A BRANCH-DIRECTED DATA CACHE PREFETCHING TECHNIQUE FOR INORDER PROCESSORS

A Thesis

by

REENA PANDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Engineering

A BRANCH-DIRECTED DATA CACHE PREFETCHING TECHNIQUE FOR INORDER PROCESSORS

A Thesis

by

REENA PANDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Co-Chairs of Committee,	Paul V. Gratz
	Jiang Hu
Committee Members,	Eun Jung Kim
	Deepa Kundur
Head of Department,	Costas N. Georghiades

December 2011

Major Subject: Computer Engineering

ABSTRACT

A Branch-directed Data Cache Prefetching Technique for Inorder Processors.

(December 2011)

Reena Panda, B.Tech, NIT Rourkela, India

Co-Chairs of Advisory Committee: Dr. Paul V. Gratz
Dr. Jiang Hu

The increasing gap between processor and main memory speeds has become a serious bottleneck towards further improvement in system performance. Data prefetching techniques have been proposed to hide the performance impact of such long memory latencies, but most of the currently proposed data prefetchers predict future memory accesses based on current memory misses. This limits the opportunity that can be exploited to guide prefetching.

In this thesis, I propose a branch-directed data prefetcher that uses the high prediction accuracies of current-generation branch predictors to predict a future basic block trace that the program will execute, and issues prefetches for all the identified memory instructions contained therein. I also propose a novel technique to generate prefetch addresses by exploiting the correlation between the addresses generated by memory instructions and the values of the corresponding source registers at prior branch instances. I evaluate the impact of the prefetcher by using a cycle-accurate simulation of an inorder processor on the M5 simulator. The results of the evaluation show that the branch-directed prefetcher improves the performance on a set of 18 SPEC CPU2006 benchmarks by an average of 38.789% over a no-prefetching implementation and 2.148% over a system that employs a Spatial Memory Streaming prefetcher.

To my family

ACKNOWLEDGEMENTS

I am extremely thankful to my advisor, Dr. Paul V. Gratz, for giving me an opportunity to work under him. His constant guidance and support always helped me move in the right direction in my research. I am grateful to him for making my research experience a memorable one, and my respect goes to him.

I would like to express my gratitude to my committee members Dr. Deepa Kundur, Dr. Eun Jung Kim and Dr. Jiang Hu for agreeing to be on my thesis committee and providing me valuable feedback.

I would also like to express my sincere thanks to Dr. Daniel Jimenez, for taking keen interest and giving his invaluable advice and suggestions during the course of this work.

Special thanks to Ehsan Fatehi for his help with the M5 simulator and other members of my research community for their support, advice and suggestions over the past one year.

My time at Texas A&M was made enjoyable in large part due to many friends that became a part of my life. I am grateful for the time spent with roommates and friends.

I would also like to thank my family for their love, support and encouragement all through my life. Lastly, I would like to thank god for blessing me with this opportunity and giving me the strength to reach this point.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Owing to significant micro-architectural advancements as well as technology scaling, the performance of microprocessors has improved at a tremendous pace over the past couple of decades. However while the processing speed has increased significantly, the memory access speed has not scaled accordingly. So, the memory access latency is becoming a serious bottleneck towards further increase in system performance.

Many memory latency hiding techniques have been proposed in the literature so far in order to reduce this growing gap between memory and processor speeds. One such technique is the use of “caches” [1]. A cache is a smaller and faster memory that resides between the CPU and the main memory and thereby, allows faster access to data that resides in it. It basically exploits two important characteristics of programs, namely, spatial and temporal locality. It does so by storing, the recently used/demanded data (thereby, exploiting the temporal locality) and the data that resides closer to other demand-fetched data in the memory (thereby exploiting spatial locality). The idea is that such data have a greater chance to be accessed by the CPU than others. As long as these characteristics hold true, complete memory accesses can mostly be avoided, thereby providing performance benefits. Several enhancements have also been proposed to the cache implementation and handling, like lock-up free caches [2], better insertion and replacement algorithms etc. However, even with all these advancements, a single cache miss through all levels still causes a loss of more than hundreds of processor cycles and is thus, detrimental to system performance.

This thesis follows the style of *IEEE Transactions on Automatic Control.*

Another technique that has been widely adopted to hide long memory latencies and also to exploit instruction level parallelism is out-of-order execution. Among its other benefits, out of order execution allows instructions, following a long latency missing instruction, to execute, being constrained only by the true data dependences or the size of the instruction window. Thus with out-of-order support, it becomes possible to overlap memory accesses with actual execution, thus hiding some of the penalties of a complete memory access. But as the technology is gradually moving into the submicron realm, superscalar processors (which are capable of supporting out of order execution) are becoming increasingly expensive to implement. This is because such processors employ several complex hardware units like the Reorder buffers, issue and wake up logic, multi-entry buffers etc., which are very power hungry and also have higher area requirements. These concerns have therefore, started diverting the attention back to the simple inorder processors, which have lesser power and area requirements.

A third technique that allows hiding memory access latency is prefetching. Prefetching predicts the data that will be used by the processor in future and generates requests to bring them closer to the processor before an actual request is sent out for them. So, if the prediction turns out to be correct, the demand request gets satisfied in the cache and the need to fetch the data from main memory is eliminated. But, like any other speculative technique, prefetching is not perfect and hence, it is likely that many prefetched blocks may be either useless or ineffective. However, such prefetched data may still evict more useful data from the cache and hence, can cause cache pollution. Additionally, a large number of prefetch requests sent to the main memory may impact the limited available bandwidth and hence, can cause delay in servicing other demand requests.

I.1 Thesis Statement

This thesis proposes a data prefetching mechanism as a means to reduce the impact of long memory latencies on system performance. This proposed scheme takes advantage

of the high prediction accuracies of current-generation branch predictors to accurately generate a future basic-block trace of the program and then, issues prefetches for all the identified memory instructions in these basic blocks. In addition, this thesis describes a novel technique to capture the data access behavior by observing the runtime modifications to the register values that used for memory address computation. The goal of this thesis is thus, to demonstrate that: a) the behavior of control instructions can be efficiently exploited to enable timely and effective prefetching and b) data addresses can be accurately predicted by monitoring the runtime updates to the address-generating register values.

I.2 Thesis Contributions

In this thesis, we propose a data prefetching technique so as to bridge the growing gap between processor and memory speeds and thereby, leading to performance benefits. While most of the existing prefetchers predict future accesses based on current memory misses, our prefetcher leverages the high prediction accuracies of current-generation branch predictors to accurately generate the future basic block trace that the program will follow and initiates data prefetching much before the actual execution of the instructions in the corresponding basic blocks begins. Our proposal is based on the idea that branch instructions determine the execution path of any program, i.e., which basic block of instructions gets executed and in what sequence is determined by the direction of the branch instructions contained in the path. Different basic blocks tend to operate on same/different data and contain instructions to operate on data in a particular pattern. So, given that branch instructions determine which basic blocks would get executed in any instance of the program run, the access pattern of data that is manipulated in those basic blocks can also be linked to the prior branch instructions.

We build our system based on the observation that the address values generated by the memory instructions in a basic block are quite predictable even at an earlier branch

instruction. We establish this correlation in hardware, by associating the source register indices being used for address computation by the memory instructions in any basic block to their preceding branch instruction (the entry point of the block). By making use of the actual register values at that execution instance and not data access history, we can prefetch even those instructions that do not exhibit regular strided access patterns, but still generate predictable address values starting from the dynamic register values.

In this thesis, we propose a practical hardware design of our data prefetcher for an inorder processor implementation. Due to their lower power and area requirements as compared to their superscalar counterparts, inorder processors have been receiving a lot of attention lately. They are thus making their way into mainstream multiprocessor and multi-core based designs. Many modern processors like Intel's Atom processor [3], Sun’s UltraSPARC T1 “Niagara” [4] have preferred to incorporate a number of smaller inorder cores over larger superscalar cores, thereby saving power and area. However, inorder processors have reduced single-threaded performance. The reason is partly because they allow very limited execution around the data cache misses. So, techniques like prefetching become more important for such architectures, as a means to bridge the growing gap between processor and memory speed. It is however important to note that, this prefetcher design is not architecture-specific and can be implemented with any processor architecture.

Finally in this thesis,

- We demonstrate that data addresses generated by memory instructions are predictable at prior branch locations by exploiting the runtime values of those registers that are used for memory address computation.
- We propose a practical hardware implementation of a prefetcher for the L1 Data Cache that allows look ahead across basic blocks and exploits the above-mentioned correlation to initiate prefetching.

- Branch directed prefetcher provides a mean speedup of 38.789% over a baseline system with no prefetching. While the Spatial Memory Streaming (SMS) [5], [6], one of the best performing practical prefetcher, provides a mean speedup of 35.87% over the baseline. Our final implementation also provides an IPC improvement of 2.14 % over SMS.
- We also discuss several enhancements to the base prefetcher design to improve the performance and accuracy of the prefetcher.

I.3 Thesis Organization

This document is organized as follows. Chapter II gives an overview of the proposed approach and discusses the motivation behind the same. In Chapter III, we provide an overview of the prior work in the areas important to this thesis. Chapter IV presents a detailed description of the system architecture. In chapter V, we discuss our simulation methodology and evaluate the results. In Chapter VI, we discuss few observations that were made, while implementing the different design alternatives. Finally, Chapter VII concludes this thesis and discusses future work.

CHAPTER II

BACKGROUND AND MOTIVATION

This chapter provides a general overview of the branch-directed prefetching system and also discusses the motivation behind the proposed approach.

II.1 Background

The direction taken by the control instructions determines the execution path of any program. In other words, which basic block of instructions gets executed and in what sequence is determined by the direction of execution of the control instructions encountered along the path. In this thesis, it is claimed that since branch instructions control the execution path, the data access patterns of subsequent basic blocks could also be dependent on/linked to the previous branch behavior. For example, consider a "C" code fragment comprised of an if-else code block (see Figure II.1).

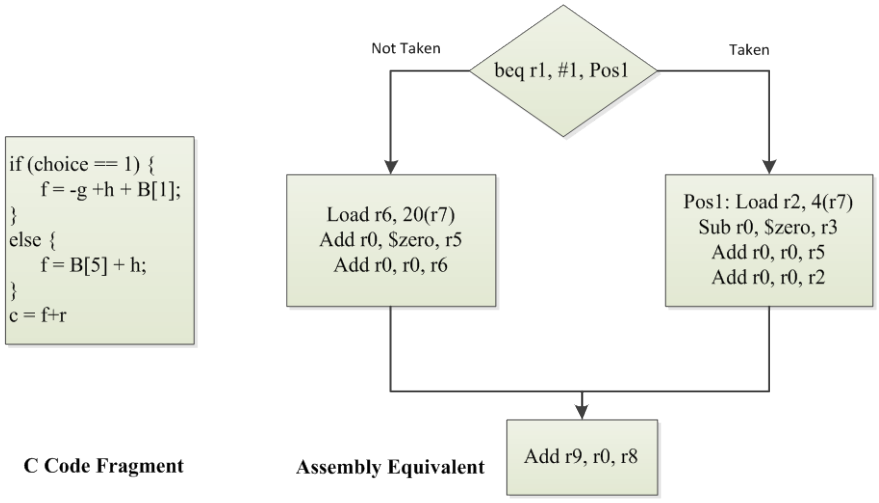


Figure II.1 An example illustrating dependence between branch instructions and data access patterns

The block of code (if-block or else-block) which gets executed following the control instruction depends on the direction taken by it. So, the data that is going to get requested in the future execution phase and its access pattern is also dependent on the branch instructions encountered along the path and their direction of execution. Given that such correlation can be established, it implies that data prefetching can be initiated at the decode time of the branches, without waiting for the corresponding memory instructions to start executing. Additonally, data prefetching can be initiated even earlier by employing a reasonably accurate fast forwarding scheme that can predict the future execution path following a branch instruction. This implies that while one branch instruction is being executed, the future path of execution can be predicted therefrom and then, prefetches can be issued for those memory references that are linked to the future branch instructions contained in the path. Also, if the path prediction accuracy is not very high, then a confidence estimator can be employed to prevent speculating too deep along a wrong path (if at all) instead of allowing the lookahead to continue as long as possible. The process adopted to enable branch-directed prefetching is shown schematically in Figure II.2.

This thesis proposes a data prefetcher that establishes correlation between the memory instructions used in a basic block and their prior branch instructions. Later, it employs a lookahead scheme to predict the future path of execution and exposes the memory instructions identified along the path. Unlike prior works in this area, which mostly find correlation among the actual data addresses used by the instructions at consecutive execution instances, we propose to associate register indices being operated by the memory instructions (as source registers to generate data address) to their preceding branch instructions (the entry points of the basic block) and use this correlation to guide prefetching. This idea is based on the premise that register values at the time of data address generation would not be very different from their corresponding values at a time when the preceding branch instruction was executed. By exploiting such register-based correlation, the branch-directed prefetcher can not only predict data addresses which

display a regular strided-access pattern, but also can take advantage of the dynamic values of the registers at run-time to predict irregular and isolated data accesses.

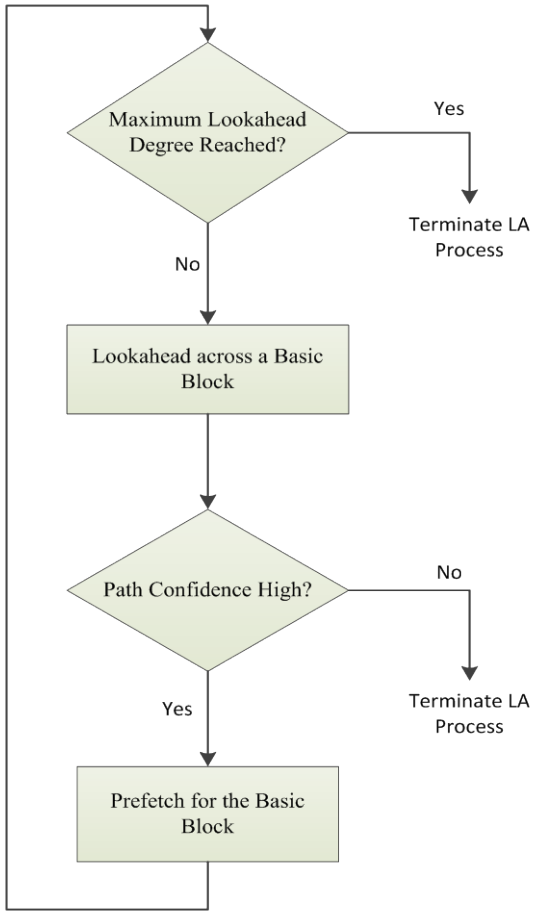


Figure II.2 Flow chart showing the Branch-Directed Prefetching Algorithm

In order to determine if such correspondence exists, we conducted an experiment to demonstrate the degree of correlation between the data addresses generated by memory references and the corresponding register values at prior branch instructions. Before describing the details of the experiment, the meaning of certain terms are clarified first, which have been used throughout this thesis. A "***spatial region***" is defined as a coarser unit of memory, consisting of multiple consecutive cache blocks [5], [7]. Based on the above definition of the spatial region, two address values are said to be "***correlated***" if they fall into the same spatial region.

The experiment was conducted using application traces (corresponding to first 300 million committed instructions) collected from a subset of SPEC CPU2006 benchmarks. The traces consisted of: a) a dump of the architectural register file at each branch location, b) the effective addresses generated by each subsequent memory instruction, c) the corresponding source register indices used for address computation by the memory instructions. An offline analysis was then, performed on these traces to find out the degree by which the register values at prior branch instructions are correlated to the actual effective addresses generated by the instructions in those basic blocks. Finally, the percentage of memory instructions which demonstrated this correlation (where correlation implies falling into the same spatial region) with their preceding branch instructions was recorded. Also, the impact of the assumed region size on the degree of correlation was monitored. Results of this experiment for two different region sizes (512 Bytes and 256 Bytes) are shown in Figure II.3.

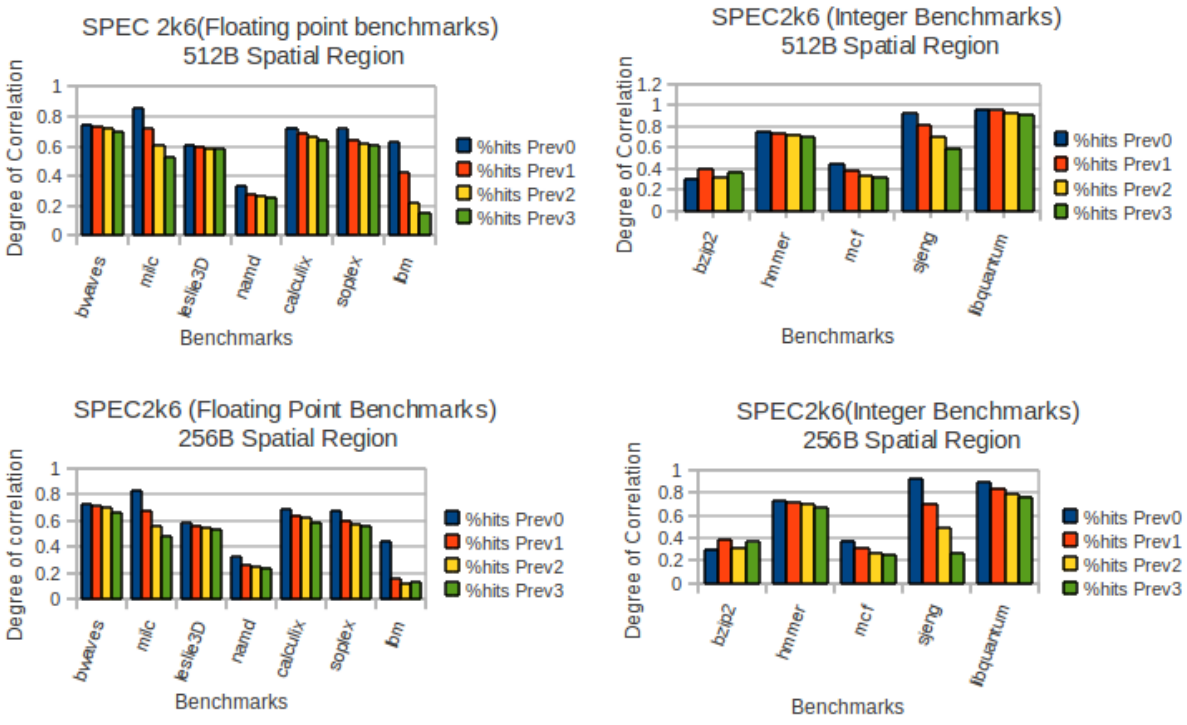


Figure II.3 Graphs showing the degree of correlation between generated data addresses & corresponding source register values at a prior branch

Corresponding to the figure, Prev 0 implies the case when the memory instructions were compared with their immediately preceding branch instruction, Prev 1 corresponds to the case where memory instructions were compared with the branch preceding their immediately preceding branch instruction and so on.

From the figure, we can observe that most of the benchmarks exhibit significant degree of correlation between the data addresses generated by memory instructions and corresponding register values at previous branch instructions. Also, as expected, the correspondence is stronger with respect to the immediately preceding branch instruction and it gradually reduces as the correlation is tested with older branches in program order. This is because greater is the distance between a memory instruction and the branch instruction in question, higher is the chance that other register-defining instructions would modify the value of the register in between. Also, another important point to note is that percentage of correlation reduces with the size of the recorded spatial region.

Results of this experiment motivate the idea of prefetching all the cache blocks contained in the spatial region that holds the address given by the register value at a previous instance. Also, a region size of 512 bytes is chosen for use in all our experiments (applicable only in non-loop mode of operation, refer Subsection IV.1) in this thesis. The evaluation and analysis of the impact of varying region sizes on the performance of the prefetcher is left for future work.

It is obvious that greater is the correlation of memory addresses to prior branch instructions, better is the opportunity to look ahead across deeper basic blocks and be able to issue useful prefetches. But on the whole, this experiment demonstrates that prefetches can be issued with a certain degree of accuracy, for memory instructions at prior branch instructions by only using the values of the corresponding source registers at that instance. It is however to note that more correlation can be exploited, than

demonstrated in this experiment if the difference (offset) between the two compared values can be estimated or predicted.

II.2 Motivation

In this section, we will summarize the motivation behind the adopted approach (as also discussed in the previous section) with a theoretical example. Figure II.4 is a code snippet from the leslie3D benchmark from the SPEC CPU 2006 benchmark suite. The dynamic program sequence consists of 3 basic blocks. In the first basic block, there are 2 memory instructions, sequence number (SeqNum) 2 and 3. Since no other register-defining instructions exist between SeqNum 2,3 and the branch instruction at SeqNum 1, that can change the values of registers R30 and R3, the address going to be generated at SeqNum 2 and 3 is predictable at instruction 1, if the register values R3 and R30 are known.

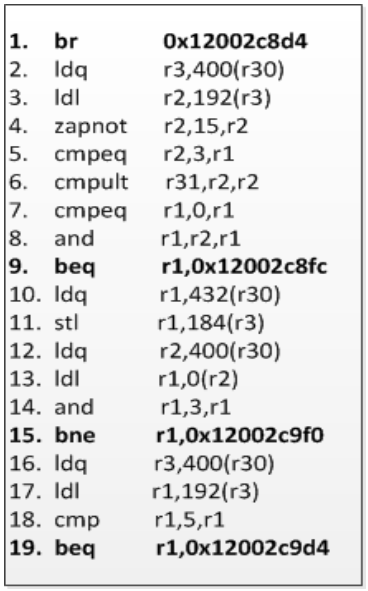


Figure II.4 Code fragment from Leslie3D benchmark (SPEC CPU2006)

Similarly, in the second basic block, there are 4 memory instructions: SeqNum 10, 11, 12 and 13. SeqNums 10 and 12 use R30, SeqNum 11 uses R3 and SeqNum 13 uses R2

as their source registers for address generation respectively. It can be seen that none of the instructions contained in Basic Block 1 define the value of R30, which implies that the data addresses generated at SeqNum 10 and 12 in basic block 2 are quite predictable at SeqNum 1. However, the two memory reference instructions, SeqNum 2 and 3 define R2 and R3 respectively in basic block 1. Hence, the value of R2 and R3 at SeqNum 1 is more likely to be different from the addresses generated at SeqNum 11 and 13 respectively. But it is note that the address values are still predictable at the branch instruction corresponding to SeqNum 9. Note that, in our scheme a significant fraction of this variability can be captured by prefetching the entire spatial region around the runtime register values.

Another factor that motivates exploiting branch-directed correlation to guide prefetching is that any typical program will have less number of control instructions as compared to the number of memory instructions. So, this approach should theoretically, need much smaller predictor table sizes than most of the prior prefetchers that establish memory instruction-based correlation to capture the same information.

CHAPTER III

PRIOR WORK

This chapter reviews some related concepts before embarking on the specifics of the thesis. Subsection III.1 discusses the different data prefetching techniques that have been proposed in literature and compares our proposed solution against a few. As discussed in the previous chapter, our prefetcher employs a confidence estimator that estimates the likelihood that our execution path prediction is correct in order to limit prefetches along a wrong path of execution. Subsection III.2 reviews the importance of such confidence estimation techniques in general and also, discusses few branch confidence estimation techniques that have been previously proposed in literature.

III.1 Data Prefetching Techniques

Data Prefetching techniques have been explored extensively as a means to tolerate the growing gap between processor and memory access speeds. Broadly, the proposed solutions in this area can be classified as hardware-driven or software-directed techniques. ***Software prefetching*** [8], [9], [10] schemes perform static compile-time analysis of the likely memory accesses to learn patterns and predict future prefetch candidates. ***Hardware prefetching*** schemes, on the other hand use dynamic runtime information and thereby, issue prefetches far in advance so as to mask the off-chip latencies. This section illustrates few hardware-directed prefetching schemes that have been proposed in literature so far.

III.1.1 Sequential Prefetching

Sequential Prefetching [11] is one of the simplest hardware prefetching schemes. It proposes prefetching the successive cache blocks that follow a currently

accessed/demanded block. Several variations have been proposed to this basic scheme, which includes what type of accesses to a block initiate a prefetch (giving rise to the Prefetch-on-hit or Prefetch-on-miss schemes) and the number of blocks that are prefetched per access to a cache block (basically, the degree of prefetching). Tagged prefetching is another variation of this approach, where prefetching is initiated both on a cache miss as well as on a prefetch hit.

III.1.2 Stride Prefetching

Stride Prefetching [12] involves monitoring the patterns of memory accesses generated successively by memory instructions, with an objective of identifying constant-stride references which are typical of loop-based behavior. In order to achieve this, the stride prefetchers maintain a table structure that gets indexed with the memory instruction PC and contains the last address referenced by that instruction, the established stride, and a finite state machine that guides the prefetching scheme. This scheme is very effective for applications which are loop-based and demonstrate a very regular access pattern. However, for the general class of applications, which do not always exhibit regular strided memory access patterns, this scheme cannot provide much performance benefit.

II.1.3 Pointer-based Prefetching Techniques

C***ontent-directed prefetching*** (CDP) technique was proposed by Cooksey et al. as an effective prefetcher for the pointer-intensive applications [13]. It basically examines each address-sized word of the fetched or subsequently prefetched data in order to find likely pointer addresses and then, it initiates prefetch requests for those data that are identified as potential addresses. As a result of its aggressive policy, CDP has the potential to run many instances ahead of the current execution sequence and prefetch data, pointed by likely pointer addresses, into the cache. Its other advantages are that it does not require any state information and also does not require any training. However, because of its

aggressive nature, it tends to generate a lot of useless prefetches.

In an attempt to minimize such useless prefetches, Mutlu et al. proposed an enhancement to the basic CDP implementation, by adopting a combined hardware/software approach [14]. In this modified scheme, the compiler provides hints to inform the hardware about which pointer addresses would be useful over others. They also proposed a hybrid prefetcher implementation where the CDP is used in conjunction with a stream prefetcher and then, runtime feedback information is used to manage the interference between these two classes of prefetchers.

III.1.4 Runahead Mechanisms

Runahead-based prefetching schemes are based on the idea of pre-executing a set of instructions speculatively following a long latency operation, like an L2 cache miss and then, using the results obtained during that process to initiate prefetching. In the subsequent paragraphs, few such techniques are reviewed.

One of the earliest works on runahead prefetching was proposed by Dundas and Mudge [15]. In the paper, the authors proposed a data prefetching mechanism that generates addresses based on the results of pre-executing future instructions under a cache miss. Two approaches are proposed to realize the prefetcher: - a) a conservative approach in which instructions are not executed speculatively beyond branch instructions while in the runahead mode and b) an aggressive approach, in which branches and jumps are assumed to be correctly resolved during runahead. This scheme requires an extra check-pointing register file to save the architectural state before entering the runahead mode and makes use of the idle execution unit to facilitate runahead during the long latency data miss. But this method adds to the miss latency overhead by requiring to checkpoint the main register file during every data miss and restoring the checkpoint on the completion of the miss.

Mutlu et al. proposed an implementation to support runahead execution in out-of-order processors [16]. Their system is also based on entering a runahead mode post a long latency memory miss, when the future instructions get speculatively pre-executed and the corresponding results are used to initiate prefetching. Though this system is quite effective in the event of L2 misses, it suffers from a few drawbacks. Firstly, there is a large overhead in restarting normal execution after restoring the checkpoint, when the miss returns. Also, because of this overhead, the effectiveness of this mechanism to handle shorter latencies like the L1 cache miss latencies is reduced. Additionally, since the same hardware is used for runahead mode execution, computation cannot be overlapped with an L2 miss.

Finally in most of the runahead proposals, in an attempt to minimize hardware overhead, the prefetching opportunity gets confined to finding idle execution slots or idle context in a multithreaded environment.

In our current proposal, we also attempt to prefetch ahead of the currently executing basic blocks. But instead of relying on pre-executing the instructions following a long latency event, we make use of modest hardware to establish and exploit the dependence between memory references and their prior branch instructions. Additionally, the runahead mechanism relies on misses to initiate prefetching, but our approach tries to avoid the first misses as well. Moreover, unlike runahead mechanism, our approach is completely transparent (non - intrusive) to the execution in the main pipeline and does not add any additional overhead to the miss-handling latency.

III.1.5 Region Based Prefetching Techniques

Another technique that has been explored to improve prefetching performance is exploiting spatial locality over larger areas in memory, bigger than a single cache line. These approaches see a coarser view of memory, generally made of a few contiguous

cache blocks (called a spatial region) and try to find correlation or access patterns with respect to this coarser view. Few such techniques that exploit region-based correlation to enable prefetching are described in the following paragraphs.

Spatial Memory Streaming (SMS) is a spatial-region based prefetching proposition by Somyogi et al. [5]. It is one of the best-performing prefetchers proposed in literature currently. SMS makes use of code-based correlation to take advantage of spatial locality over larger regions of memory (called spatial regions) in the applications. As an application runs, SMS records access patterns over spatial regions in the form of bit vectors, over a period of time called the spatial region generation (defined as the time from when the first block of this region was brought into the cache till when an accessed block gets evicted). At the end of a spatial generation, these recorded bit patterns are transferred to a pattern history table (PHT). In their work, the authors show that an indexing mechanism that combines the PC and the initial missing offset into the region gives better results over other indexing schemes. But one potential issue with SMS is that it cannot predict the first misses into a region. To overcome this disadvantage, Somyogi et al. proposed an extension to SMS called Spatio-Temporal Memory Streaming (STEMS) [17]. STEMS exploits temporal access characteristics over the larger spatial regions and finer access patterns within each spatial region to re-create a temporally ordered sequence of misses and prefetches for the same. By employing both temporal and spatial characteristics, it improves the performance by 3% over the SMS scheme. However, this performance benefit is achieved at the expense of a huge hardware overhead (in the order of several megabytes), which makes this design slightly impractical to implement currently.

III.1.6 Branch-directed Data Prefetching

This is another class of prefetching that exploits the relationship between branch instructions and subsequently following memory instructions to identify prefetch

candidates. Although branch-based correlation has mostly been explored in the instruction-prefetching domain, there has been some work [18], [19] that applies the same to solve data prefetching issues**.** The branch-directed data prefetchers are based on the idea that since branch instructions control the execution path through a program, data accesses in the subsequently following instructions are also dependent on their behavior and hence, can be linked to them. However, in most of the proposed approaches, memory reference instructions are directly correlated with prior branch instructions and then, some variant of stride-directed scheme is used to guide prefetching. This section discusses some major work in this area.

The earliest work on branch-based data prefetching [18] associated the history of data references to the previous branch instructions in the Branch Target Buffer (BTB). Each BTB entry is extended to contain the last accessed data address field, a stride field and a 2-bit counter to handle the finite state machine to enable stride prefetching, corresponding to each memory instruction. Equipped with all this state, the BTB is then used to issue prefetches for load instructions following the branch instruction in the program flow. Thus, when a branch instruction gets decoded, the corresponding BTB entry is looked up to find the possibility of a potential prefetch. In case such an opportunity exists, a prefetch address is generated by adding the currently accessed data address to the estimated stride, in advance of the actual issuing of the loads.

Pinter and Yoaz proposed another branch-directed prefetching data scheme called the ***Tango prefetcher*** for superscalar implementations [19]. The authors propose their solution again, as an enhancement over the stride-based reference prediction table approach suggested by Chen and Baer [20]. To issue prefetches fast enough to benefit a superscalar implementation, a lookahead scheme is employed that allows jumping from one branch instruction to another in a single clock cycle. Prefetches are then, issued for the memory instructions linked to the looked-ahead branch instruction using a modified version of the stride prediction table. In order to limit the impact of prefetching on

demand cache access behavior, Tango issues prefetch requests only during idle time slots and hence, it does not overload the cache ports. However, this system has certain limitations. Firstly, Tango is based on a modified stride prediction algorithm. Hence, the opportunity to prefetch is confined to those data structures that have uniform strided access patterns and hence, the general class of applications cannot be benefited from the same. Secondly, in Tango, once the lookahead process starts, it is allowed to proceed till a misprediction is detected in the main execution pipeline. Hence, owing to the imperfect branch prediction accuracies, the lookahead scheme is very likely to go deeper along a wrong path of execution and thereby, issue many useless prefetches.

Table III.1 shows the hardware overhead and performance benefits of previous branch-directed prefetcher implementations.

Table III.1 Hardware overhead and performance benefits of prior branch-directed prefetchers

Prefetcher	Architecture	Hardware Overhead	Speedup over Baseline
Branch-Directed and Stride Based Prefetcher[18]	Inorder processor	1024 entries in BTB, unlimited linked data entries per BTB entry	Approximately 4% improvement in data cache hit rate
Tango Prefetcher[19]	4-wide superscalar processor	Approximately 4.5KBytes	Average speedup = 1.36

In this thesis, we also employ a lookahead scheme to generate timely prefetches similar to that adopted in Tango. However, our system has certain advantages over the previously proposed branch-directed schemes including Tango. Our scheme enables prefetching by exploiting the correlation between the values of the source registers (that are used for memory address computation in basic blocks) at prior branch instructions and actual addresses generated by the corresponding memory instructions. This approach has many benefits over exploiting only memory instruction-based correlation. One such

benefit can be realized by examining the following code fragment.

```
Loop: 1. stt r1, 16(r2)
      2. stt r1, 24(r2)
      3. Lda r2, 16, r2
      4. Cmpeq r2, r1, r3
      5. Br Loop
```

This program sequence is taken from the Leslie3d benchmark from the SPEC CPU 2k6 benchmark suite. Since this is a loop-based code fragment, both the stride-based techniques (like Tango) and our current implementation can accurately prefetch for the same. But it is interesting to note that both instructions 1 and 2 manipulate the same register (r2) for their address computation, even if they are different instructions. In such a case, a stride-based prefetching scheme needs to save two separate entries for these two instructions (1 & 2) to accurately prefetch for them. But our register-index based prefetching scheme can save the same amount of information using a single entry corresponding to register index r2 (linked to the branch instruction, 5). Another advantage of our technique is that by exploiting branch-register correlating links, the dynamic runtime values of the registers can be used to enable prefetching for even those data that show irregular memory access patterns. But the previous methods can only take advantage of strided memory access patterns.

Having discussed the previously proposed data prefetching strategies, we next provide some background about confidence estimation mechanisms. Such techniques help to estimate the confidence of a certain prediction and hence, are used to limit the degree of speculation. We also employ a similar technique in our current scheme to enhance the accuracy of our prefetcher by limiting prefetches along an incorrectly predicted path of execution.

III.2 Confidence Estimation Techniques

Confidence estimation is a micro-architectural technique that allows control over speculation by predicting whether the speculation will be correct or not, before the actual outcome is established / known. Such techniques can be applied in areas like branch prediction, prefetching etc. For example, in the context of branch predictions, a confidence estimator can be used to classify the dynamic predictions made by a branch predictor into high confidence or low confidence categories.

In our current work, we are more concerned about estimating the "***execution path confidence***". Unlike branch confidence, a path confidence estimate measures the confidence that a predicted execution path will be actually followed. Such a path may span multiple basic blocks and hence, can be used to limit lookahead from proceeding deeper along a wrong path of execution. Many path confidence estimators have been proposed in the past. But mostly all such approaches are based on the idea that since the control flow instructions determine the execution path, branch confidence estimators can themselves, be used to derive the path confidence estimate fairly accurately. In this subsection, a few branch-based and path-based confidence estimators are reviewed, that have been proposed in literature.

Jacobsen et al. proposed an accurate confidence estimation mechanism (called the ***JRS confidence estimator***), aimed at controlling the branch-prediction based speculation [21]. It is based on the idea that a very small subset of static branches causes a majority of dynamic mis-predictions and that most mis-predictions occur in clusters. Hence, in their approach, it is attempted to identify those branches that were mis-predicted in the recent past and hence are likely to mis-predict again. To identify such branches, a table of 4-bit saturating counters called the miss distance calculating (MDC) table is used, whose index is derived by xoring the branch PC with the global branch history. Each time, a branch is correctly predicted the corresponding MDC entry gets incremented and

the counter is reset to zero in the event of a misprediction. So, the table entry essentially stores the number of times a particular branch was correctly predicted consecutively in the past. Higher this counter value, greater is the probability that the prediction would be correct another time. Grunwald et al. proposed an enhancement over the JRS predictor, wherein the global history that is used to index into the MDC table also includes the prediction for the current branch in question [22]. This predictor was also shown to be better than the original JRS proposal.

Jimenez proposed a composite scheme to make better confidence estimates in relation to branch predictions [23]. For a tournament-based predictor, the author proposed to use a combination of the outputs of different confidence estimators, like the JRS, up/down and the branch predictor's self-counters to estimate a composite confidence output. Finally, whether a prediction is deemed to be of high confidence or not depends on whether the aggregate confidence estimate is above a pre-determined threshold. A variation of this technique is used for branch confidence estimation in this thesis work.

Among the efforts towards path confidence estimation, the approach adopted in many proposals is based on the idea that the higher the number of low-confidence branches along a path, higher is the likelihood of the path being incorrect. Along the same lines, the path confidence estimator records the count of the number of high-confidence and low-confidence branches encountered in a path and when the total number of low confidence branches increases beyond a certain threshold, the path is considered to be low-confident. However, this technique assumes that all low-confidence branches have the same misprediction rate and that all low confidence branches have lesser likelihood of being correct than all high-confidence ones, which may not be always true.

In contrary to the above assumption, we observed that because the branch confidence estimators are themselves imperfect, the misprediction rates observed over the different confidence categories does not correlate with the confidence value exactly. This

observation is in line with the observation made by Malik et al. in [24]. In [24], the authors do not use the value of each branch confidence category as an estimate of its correctness. Instead, they estimate the confidence value of each category based on the dynamically observed misprediction rates of the branches that fall into the same. The path confidence estimate is then, calculated by multiplying the confidence estimates of all the branches included in that path. In the same work, the authors have also presented a hardware implementation that measures path confidence using only integer addition and subtraction.

CHAPTER IV

DESIGN AND IMPLEMENTATION

This chapter presents the complete design and implementation of our branch directed prefetcher. First, a general overview of the overall system architecture is provided in Subsection IV.1, which is followed by a detailed description of the individual system components in Subsection IV.2. Finally, the chapter concludes with a working example (in Subsection IV.3) explaining how the different components work together.

IV.1 Overall System Architecture

This subsection presents an overview of the modified system architecture and discusses how the different components are tied to each other.

Figure IV.1 depicts the detailed architecture of a modified inorder core, showing the main execution pipeline as well as the additional hardware entities to realize the branch-directed data prefetcher. The additional components are as follows:

- **Branch Trace Cache (BrTc) Table**: It captures the dynamic control flow sequence of a program. It caches pairs of branch instruction PCs, where the second branch follows the first branch along a specific direction of execution of the first branch. This structure allows jumping from one basic block (defined by the entry branch instruction and its direction of execution) to the next in a single clock cycle. It thus, is used to implement the lookahead mechanism which plays a key role in making this prefetcher effective and timely.

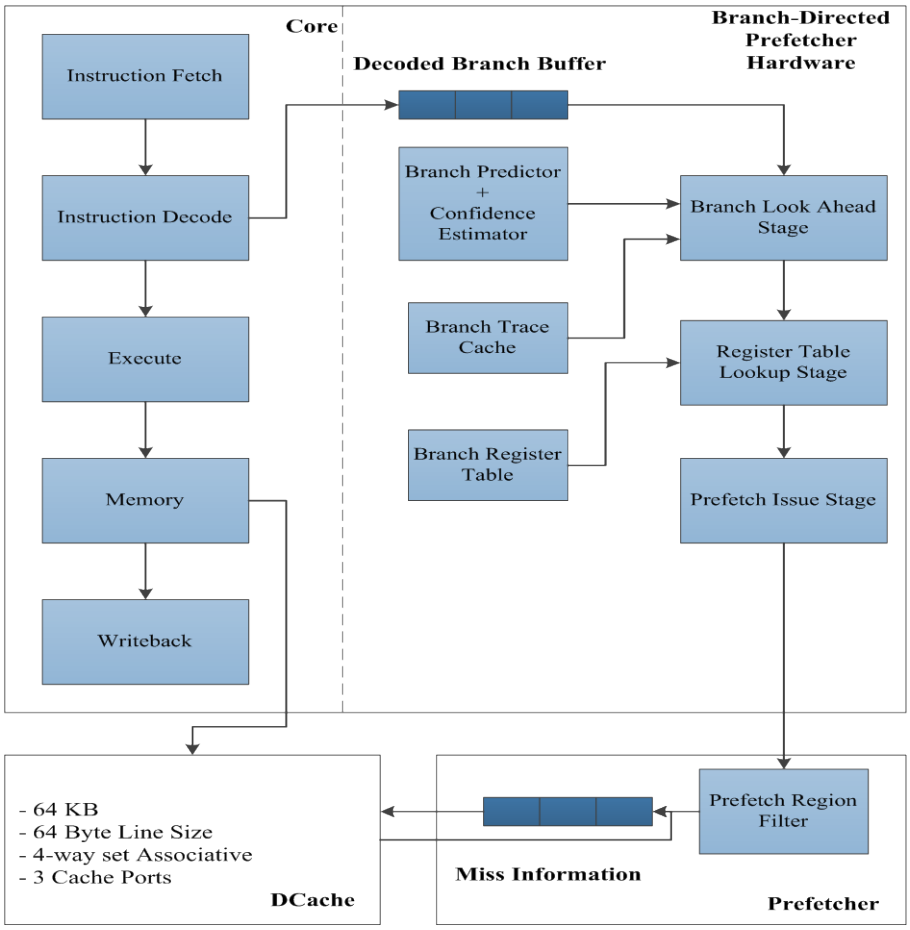


Figure IV.1 Overall system architecture

- **Path Confidence Estimator**: This component allows controlling the degree of lookahead across basic blocks, by keeping track of the confidence of the predicted execution path. As the prefetcher tries to lookahead across multiple basic blocks so as to issue prefetches for them, this unit runs in parallel and estimates the confidence that the predicted execution path will be actually followed in the main execution pipeline. Whenever the computed confidence falls below a certain threshold value, indicating greater likelihood of lookahead being along a wrong path, the lookahead process is terminated. Thus, this helps to avoid prefetching useless data, by preventing lookahead along a wrong path of execution. It is to note that this kind of control mechanism has not been explored

in any similar prior work. Note also that most of the latest branch predictors come with a built-in confidence prediction mechanism and hence, it makes the use of an additional confidence estimator unnecessary.

- **Branch-Register Table (BrReg Table)**: This is one of the most important structures towards the realization of this prefetcher. It captures the information that is used to generate prefetch addresses for future basic blocks. It links the memory instructions in any basic block to its immediately preceding branch instruction, by linking the source register indices of the memory instructions to their preceding branch instruction.

- **Prefetch-Filtering Mechanism**: Given its aggressive nature, this prefetcher tends to issue a large number of prefetches, which may not all be useful to the processor, thereby causing cache pollution. Additionally, this might also lead to increased demand on the limited bandwidth, thereby affecting performance. Hence, certain filtering techniques are in place to control the number of useless prefetches issued by the prefetcher.

As can be seen from the figure, the prefetching component is implemented as a separate pipeline referred to as the Auxiliary pipeline (A.P), parallel to the main execution pipeline. It monitors certain events of interest in the main pipeline for its functioning, but otherwise is completely non-intrusive to the actual program execution. Currently, A.P is implemented as a 3-stage pipeline where: a) the first stage is the ***"Basic-Block Look-Ahead"*** stage that allows to jump from one basic block to the next and to achieve the lookahead component of the prefetching algorithm b) the second stage is the ***"Branch-Register Table Lookup"*** stage that allows to expose memory instructions in each looked-ahead basic block & generates prefetch candidate addresses and c) the third stage is the ***"Prefetch Issue"*** stage that issues identified prefetch addresses to the prefetch queue.

As shown in the figure, the A.P is connected to the main pipeline through a 3-entry ***Decoded Branch Buffer*** (DBB). As branches get decoded in the main execution pipeline, they get inserted into the DBB (which operates in a FIFO fashion). A.P then, fetches these branch instruction PCs from the DBB and runs its lookahead algorithm to construct the future basic block trace, starting from the currently decoded branch. This is done by invoking the branch predictor and the BrTc structure repeatedly in the Basic-Block Look-ahead stage. Additionally, the path confidence estimator ensures that prefetching is allowed only along a path that can be confidently predicted starting from the current branch instruction. As the lookahead process continues, the BrReg Table structure is then invoked to identify addresses for prefetch in each looked-ahead basic block by making use of the established branch-register links. This is done in the Branch-Register Table Lookup stage of the auxiliary pipeline. The predicted addresses are passed then, through a prefetch filter to differentiate between the useful and the useless prefetches. Finally, the addresses, which are predicted to be useful, are queued up in the prefetch queue, so that they can be issued to the cache whenever there is available bandwidth and no demand requests are pending. In the current implementation, we support two modes of operation:

a. **Non-Loop Mode:** In this mode, when a prefetch address is generated, all the blocks in the spatial region containing the predicted address are issued to the prefetch queue. This is done to ensure that the variability in the address values from the past architectural register values, as a result of prefetching significantly ahead of actual execution, is taken care of. This technique also allows exploiting spatial locality in the code, if any.
b. **Loop Mode:** This mode is entered upon determination that a loop-based code is being executed in the main pipeline. In this mode, while being at one dynamic instance/iteration of a basic block, prefetches are issued for data that would be needed in a future iteration of the loop. In this mode, an entire spatial region around the predicted data is not prefetched.

By employing the lookahead mechanism, branch-directed prefetching aims to identify and eliminate as many misses as possible. But even after incorporating this prefetcher, if cache misses are encountered, then it implies that a prefetch was either not issued in a timely manner or was not accurate enough (owing to the chance of variability) or no prefetch was issued in the first place due to insufficient training of the structures. So, in such a case, the next cache line (the next-line prefetching approach [11]) following the miss block address is prefetched. The main benefit of using this combined approach is that the two techniques are complementary to each other and hence each scheme can compensate for the other's weakness, while taking advantage of the other's strengths. Next-line prefetching takes advantage of spatial locality in the application in the event of a miss. The branch-directed method can take advantage of spatial locality as long as it can predict the region of operation accurately. Additionally, branch-directed prefetching can take advantage of loop-based behavior and irregular accesses as well. So, theoretically, these two categories of prefetchers should work well together.

Finally, it is important to note that the branch-directed prefetching approach does not require any extra ports on the cache. It also gives greater preference to demand requests over prefetch requests. Details about the operation of each individual component are discussed in next section.

IV.2 System Components

This subsection describes in detail, the implementation and working of each system component, that were touched upon in the previous section.

IV.2.1 Branch Trace Cache

The first hardware component to realize this prefetcher is the Branch Trace-Cache (BrTc). As discussed before, this structure helps to capture a trace of the control flow

sequence between basic blocks by capturing the dynamic sequence of execution of branch instructions and their direction of execution. This structure allows us to look ahead across multiple basic blocks, starting from one branch instruction. It is called Branch Trace Cache because its each entry stores a trace of the executed control flow sequence.

The BrTc is implemented as a table that caches pairs of branch instructions, where the second branch follows the first branch along a particular direction of execution. The idea is that since a branch instruction and its direction of execution determines which basic block will get executed next in the program sequence, by exploiting the branch-trace cache hit information and the corresponding branch predictions, it becomes possible to jump from one basic block to another by skipping all the non-control-flow changing instructions in between. Branch Trace-Cache based lookahead approach relies on two typical program characteristics. Firstly, most instructions exhibit temporal locality. It implies that the dynamic sequences of instructions are very likely to repeat in future and hence, if they are cached, they can be used later to realize the lookahead mechanism. Secondly, branches are mostly biased towards one direction or the other. So, it is very likely that certain execution paths will be followed more frequently than others. Hence, maintaining a limited number of such paths should enable re-creation of entire program sequence (given by the combination of basic blocks) at a later point in time.

The BrTc is indexed using the current branch PC together with its predicted direction of execution, and its entries cache the next branch tag field (corresponding to the branch instruction that would be encountered if the predicted path is followed, starting from the current branch PC) and a 1-bit field to indicate if the next branch is conditional or not. A typical entry of the BrTc is shown in Figure IV.2. A branch trace cache hit requires that (1) the current branch PC matches the saved PC tag and (2) the corresponding branch prediction matches the stored direction. In case of a hit, the next basic block of execution gets exposed, which can now be used to further the lookahead process.

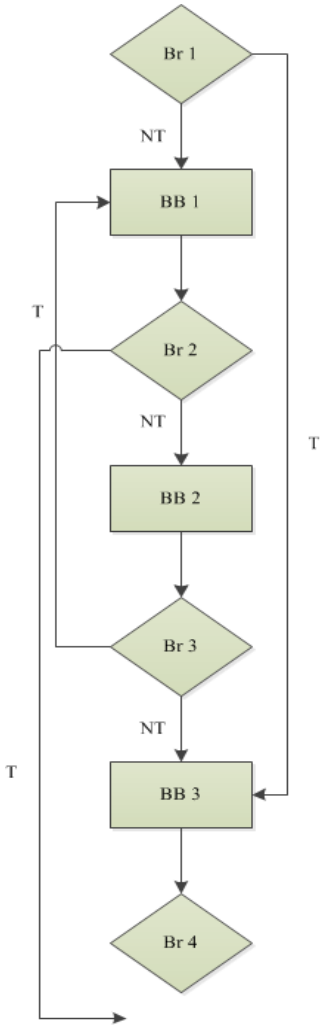
Figure IV.2 Single Branch Trace Cache entry

To enable filling the BrTc entries, two extra entities are needed, called the LastCommittedBranchInstruction (LCBI) register, which holds onto the last committed branch instruction in the main execution pipeline and the LastCommittedBranchDirection (LCBD) register, which holds onto the direction taken by the last committed branch instruction. As branch instructions commit in program order, they get linked to the branch tag saved in the LCBI register (along with the direction given by the LCBD register). To give an example of how the BrTc entries are filled and what they correspond to, consider the program sequence given in Figure IV.3 (a)**.** The corresponding control flow graph is depicted in Figure IV.3 (b)**.** In this directed graph, each bubble corresponds to one basic-block of instructions (that have exactly one entry point and one exit point) and the diamonds correspond to the branch instructions which lead into the basic blocks. The relevant filled entries of the branch trace cache for this program sequence is shown in Figure IV.3 (c).

The different design choices available for the BrTc's implementation are as follows:

a. **Table Update Policy** - BrTc entries can be trained as branch instructions get decoded speculatively or they can be filled as branch instructions retire in program order. Although the table learning time will be shorter in the first case, we choose the commit-time update mechanism in our current implementation to avoid pollution of the table by mispredicted and wrong path branches.
b. **Organization** – BrTc can be organized as a direct-mapped or a set-associative structure. Support can also be included for path associativity, which would allow simultaneous caching of multiple paths emanating from the same branch PC. Enabling support for path associativity would reduce thrashing between those

branch-pairs that start at the same address, but proceed in different directions. In our current proposal, we have implemented a direct-mapped BrTc structure, with support included for path associativity.



(b)

1. Br 1 :- bne r5, 14
2. Ld r1, 8(r2)
3. Lda r4, 8(r4)
4. St r3, 16(r4)
5. cmp r3,r2,r1

B.B 1

6. Br 2:- beq r3, 23
7. Ld r2, 0(r1)
8. Ld r4, 8(r1)
9. Lda r1, 8(r4)
10. Ld r8, 4(r1)
11. Ld r4, 64(r1)
12. cmp r4, r2, r1

B.B 2

13. Br 3:- beq r4, 2
14. Ld r1, -320(r29)
15. mul r1, r1, r1

B.B 3

16. Br4:- jmp r1
17 Ld r0, 0(r5)
18 Ld r5, 16(r5)

B.B 4

(a)

Branch PC & Direction	Next Branch	Next Branch Unconditional?
Br 1 - NT	Br 2	0
Br 1 - T	Br 4	1
Br 2 - NT	Br 3	0
Br 3 - NT	Br 4	1
Br 3 - T	Br 2	0
Br 4 - T	Br 6	0

(c)

Figure IV.3. Figure showing: (a) A program sequence; (b) Control Flow Graph of the program sequence in (a); (c) Branch Trace Cache filled state

Finally, it is to note that even if BrTc was discussed as a standalone table so far, but given its similarity to a Branch Target Buffer (BTB) structure, BrTc can be implemented

as an extension to the BTB. This will save the extra tag space to save the indexing branch instruction PC, used by the current implementation. Also, instead of saving the next branch instruction tag completely, a pointer to the next branch's position in the table [19] can be saved. These optimizations could be attempted to reduce the hardware overhead of the BrTc Table.

IV.2.2 Path Confidence Estimator

As discussed before, our lookahead logic combines the branch prediction information together with the hit information from the Branch Trace-Cache to determine a likely path of execution. But after a lookahead is initiated, some mechanism is needed to ensure that lookahead keeps proceeding along the correct path. Such a terminating condition for the lookahead process can be realized in two ways: a) the first scheme allows lookahead to proceed as deep as possible and terminates it only when a misprediction is detected in the main execution pipeline. However if the branch prediction accuracy is not very high, then it is quite likely that lookahead would proceed deeper along a wrong path fairly often. If this is not limited, a lot of data may be prefetched along a wrong path, which may lead to cache pollution and unnecessary bus bandwidth consumption. b) The second scheme employs a confidence estimation technique that limits looking ahead along low-confidence paths. This approach is conservative in nature and hence may limit prefetching opportunity in some cases, but it would control the cache pollution resulting from wrong path prefetching. Hence in this thesis work, the second approach is adopted i.e., prefetching is allowed only along those paths that can be confidently predicted starting from the current execution instance.

To estimate the confidence in the prediction of the execution path, we make use of the fact that any program contains some non-control-flow changing (ALU or memory or IO etc.) instructions and some control-flow changing instructions and that, the path followed by the program at any time depends on the direction of execution of the

constituent control-flow instructions. Therefore, to estimate the confidence of any path, it is reasonable enough to consider the confidence estimates of the constituent branch predictions alone.

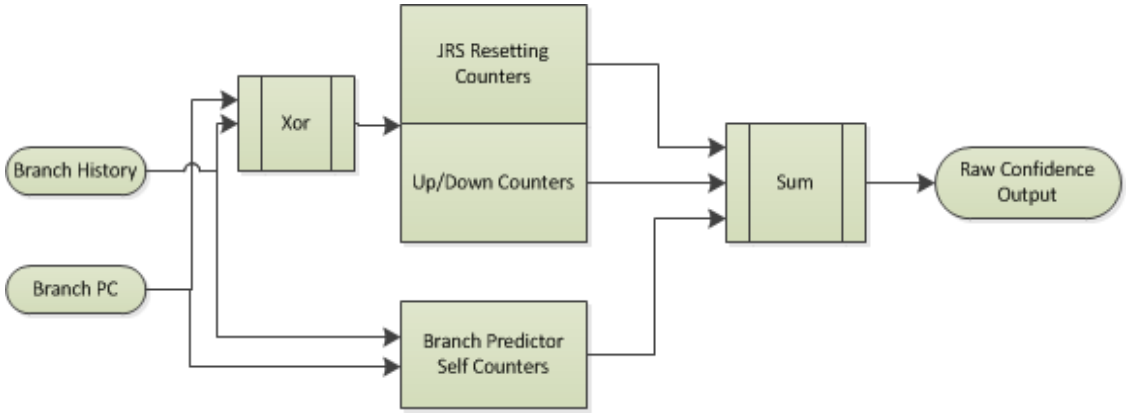


Figure IV.4 Composite Branch Confidence Estimator

To estimate branch confidence, a composite confidence estimator is employed (as suggested in [23]) that combines the JRS, up-down and self-counter based confidence estimators. This is shown schematically in Figure IV.4. The JRS and up-down estimators are arranged as a table of saturating counters that get indexed with the hash of the branch PC and the global branch history buffer. The corresponding saturating counters are incremented when a branch prediction turns out to be correct and decremented in the event of a misprediction. Therefore, to estimate the confidence of a branch prediction at any point in time, these tables is looked up using a hash of the branch PC and the global branch history buffer and the counter values are recorded. The total raw confidence value is calculated as the sum of the JRS counter value, the up-down counter and the self-counter value. We term each such raw confidence output as a "***confidence bucket***", because these values help to segregate different branch instructions into different buckets according to their predictability. In order to convert this raw confidence output into a confidence estimate, there are two possible options:

One possibility is to use a static pre-determined value for each confidence bucket. This is based on the assumption that all low-confidence branches (having low bucket values) have the same misprediction rates while, all the high-confidence ones (those having higher bucket values) are more likely to be correct. However, during our experiments, we observed that the misprediction rate of each confidence bucket does not

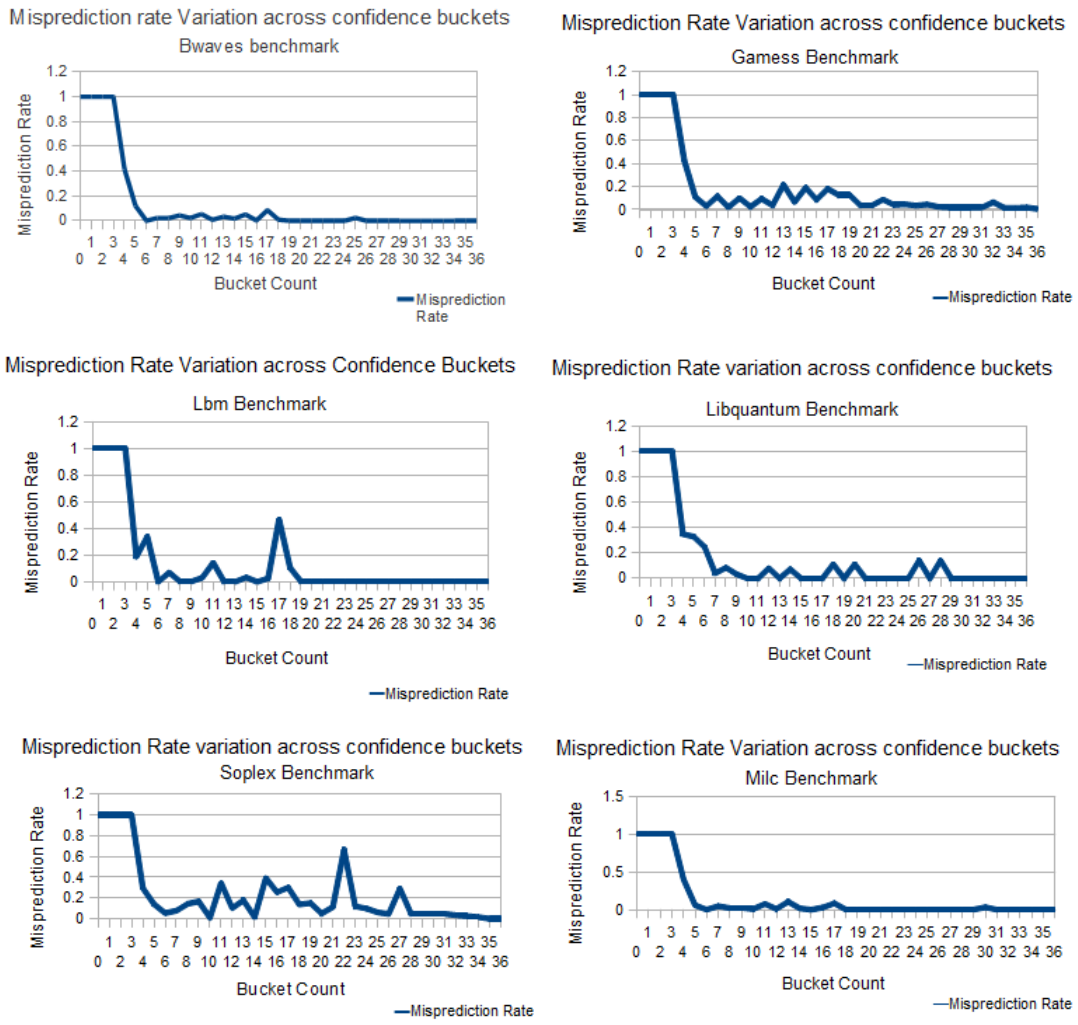


Figure IV.5. Graphs showing the variability in branch misprediction rates across the 37 confidence buckets for a set of SPEC CPU2006 benchmarks

correlate to its bucket value directly and that this trend varies with program phase as well as the application. The situation is depicted more clearly in Figure IV.5. These graphs show the misprediction rates observed across the 37 confidence buckets (JRS + Up/Down + Saturating) over a 300-million instruction run for a set of SPEC2k6 benchmarks. Misprediction rate of each bucket is calculated as the number of mispredicted branches that were predicted with that confidence value divided by the total number of branches predicted with that confidence value. We can see from the graph that many confidence buckets, which have a lower bucket number, have a better misprediction rate than many others with a higher bucket number. Similar observation was also recorded in [24].

This observation essentially, rules out the possibility of selecting a common threshold for each bucket that will hold well across all the applications and during each program phase. Hence, to take into account the variability observed across the confidence buckets, the associated confidence values are determined dynamically in our work, by monitoring the misprediction rate of each bucket. This approach is similar to that suggested in [24] except that a more fine-grained stratifier is used to filter out greater number of mispredicting branches. In this approach, counters are maintained per bucket to count the number of committed and squashed branches belonging to that category. Again, unlike [24], the confidence value of each bucket is maintained using a running estimate. Basically, the program run is divided into phases, where each phase consists of about 1/2 million branch predictions. In any program phase, the misprediction rate of each bucket is computed as the number of mispredictions falling into that category over the total number of predictions from that category. Finally, at the end of the program phase, the confidence value of each bucket is re-calculated as:

ConfidenceValue = ½ * (ConfidenceValAtTheBeginningOfInterval + ConfidenceValDuringInterval)

This running estimate (calculated as described above) gives more weightage to the misprediction rates in the latest interval, but allows for gradual changes by taking into

account the older estimates as well. Finally, the path confidence estimate is calculated as the gross product of the component branch confidence estimates.

Path Confidence = Π (Individual Branch Confidence values)

It is to note that the impact of incorporating more simplified confidence estimators has not been explored in this work. Given the other pollution controlling measures adopted in this thesis (like prefetch filtering), we think that more simplifying assumptions may be taken here without impacting performance much. Additionally, modern branch predictors come with their own self-confidence estimators and hence, do not require this separate entity to realize the branch-directed data prefetcher.

IV.2.3 Branch-Register Table

The third hardware component to realize this prefetcher is the Branch-Register (BrReg) table. This table is used to establish the links between the register indices that are used for memory address computation (source registers of memory instructions) in a basic block and their preceding branch instruction. This helps to generate data addresses for prefetching.

The BrReg Table is indexed using the Branch PC tag and the individual entries contain the registers that are linked to the corresponding branch PC and certain other fields, which are used for generating prefetch addresses. In its simplest form, a typical BrReg Entry looks as shown in Figure IV.6:

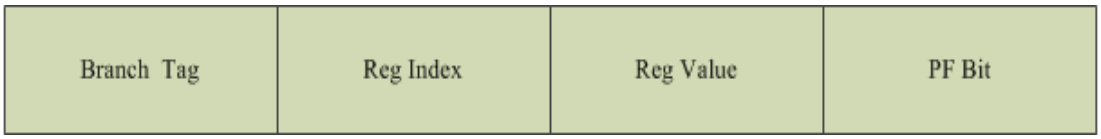


Figure IV.6 Single Branch-Register Table entry (Basic implementation)

where,

- Branch Tag field contains the Branch PC tag.
- RegIdx – This multi entry field holds the register indices, which appear as source registers for address generation in the basic block following the branch PC (given by Branch Tag field).
- RegVal – It holds the most recent value of the register, based on which a prefetch was generated during the last lookahead cycle.
- PF Bit - This is a 1-bit field and is used to distinguish between the prefetched and the non-prefetched entries. This field also helps to prevent prefetching for those basic blocks, which have already been prefetched for.

The link between memory and branch instructions gets created as the different control and memory instructions commit in program order. To establish such a link, a register called the LastCommittedBranchInstruction (LCBI) is used, which holds the last committed branch instruction in the main execution pipeline. As control instructions commit, they overwrite the existing content of the LCBI with their own PC. Hence, when memory instructions commit, they get associated with the Branch, whose PC is indicated by the LCBI register. Such links are cached in the BrReg Table. For example, for the code fragment given in Figure IV.3 (a), the corresponding learned state of the BrReg Table is given in Figure IV.7.

Branch Tag	Reg Index	Reg Value	PF Bit
Br 1	R2	-	0
	R4	-	
Br 2	R1	-	0
Br 3	R29	-	0
Br 4	R5	-	0

Figure IV.7 Snapshot of the trained Branch-Register Table

After getting trained, the BrReg table can be used to guide prefetching. This is realized as follows: To issue prefetches during the lookahead process, the BrReg table is looked up using the predicted branch PC. In case the entry is found in the table, the most recent value of the linked registers is checked in the separate register file and a prefetch is issued for an entire region (512 Byte region size) around this predicted register value. This register value is then stored in the RegVal field of the entry and the corresponding PF bit is set to 1. One important thing to note is that since this approach tries to aggressively lookahead from every decoded branch and issue prefetches for all the basic blocks that can be looked-ahead from the same, a situation may arise when consecutively decoded branches try to prefetch for the same basic blocks. While this situation is desirable in case more accurate prefetch predictions are available, but it is unnecessary when the prefetch estimate still falls into the same spatial region as the last prefetch. To avoid this situation of prefetching the same region multiple times, the following strategy is used:

When a prefetch is to be issued for a basic block, the corresponding BrReg entry (essentially the basic block) can be in two possible states:

1. The block is not prefetched yet (PF bit = 0), in which case it becomes a potential prefetch candidate immediately.
2. The block has been prefetched earlier (PF bit = 1), potentially because of a look ahead operation starting from an older branch instruction. In this case, the decision of whether to issue a prefetch or not depends on the availability of a better prediction (a different spatial region prediction). This case will arise when certain register defining instructions would have modified the value of the registers from the time the last prefetch was issued for this basic block.

Note that the above discussion described the most basic implementation of the BrReg Table. The BrReg Table can be extended to contain other information (apart from the

branch-register links) that will enable more accurate prefetching by taking advantage of different program characteristics. We discuss two such variations in this subsection, while we leave the rest for Chapter VI. Note that in all the proposed variations, the branch-register link-creation process remains the same as described previously in this subsection.

1. Offset-Based Technique

The first optimization is the result of our observation that even though, in many instances, the value of the linked register (which is assumed to be the prefetch address) at a previous branch location does not fall into the same region as the actual memory address (the static experiment also suggested an imperfect correlation), it still falls within more or less a fixed offset from it. Moreover, this offset value tends to be stable over the different dynamic run time instances of the same basic blocks. This kind of behavior may be observed because of the different addressing modes supported by the ISA (like the displacement-based addressing mode) or predictable updates taking place to the register's value within the lookahead window. For example, let us consider the code fragment as shown below:

```
1. Br1 :  beq r1, pos1
2.        add r2, 628, r2
3.        load r3, 0(r2)
4.        load r5, -512(r1)
```

In this code example, both the values of registers R2 and R1 at Br 1 (SeqNum 1) would not exactly match with the memory addresses generated at SeqNum 3 and 4 respectively. This is because SeqNum 3 is preceded by an instruction that re-defines the value of R2. Similarly, SeqNum 4 uses the displacement addressing mode and hence, the value of R1 at Br 1 would be different from the address

generated at SeqNum 4. But, it is interesting to observe that in almost all dynamic instances of this basic block, the value of R2 at the branch position will be offset by 628 as compared to the address generated by SeqNum 3. And the value of R1 will similarly be offset by a value equal to 512. To take advantage of such cases, we add another field to the Branch-Register table called the “Offset” field. So, the modified BrReg table looks as shown in Figure IV.8:

Figure IV.8 Single Branch-Register Table entry (Offset implementation)

In this new implementation, each entry gets extended to include two additional fields: - Offset - This field holds the difference between the register value at the preceding branch instruction and the actual address generated at the memory instruction, using this register as the source index.

- SeqNum field - This field holds the last few bits of the sequence number of the branch instruction which had initiated the look-ahead process. This field ensures that the offsets are set by only those instructions which have a greater sequence number than the branch in question, i.e., it occurs later in program order.

The flow chart explaining the procedure to generate prefetch addresses by using the modified arrangement is shown in Figure IV.9 (a). To generate data addresses for prefetch, the look-ahead Branch PC is used as a tag to look-up into the BrReg table. Prefetching can only be initiated if the entry is found and is in an unprefetched state or if a better prediction is available for an entry that is already in the prefetched state. The address for prefetching is calculated as the sum of the actual register values and the offset field (if any). Note that a prefetch is issued to

an entire region around the address computed above. The entry's corresponding PF bit is also set to 1 to avoid future prefetches to the same basic block and the generated address value is saved in the RegVal field.

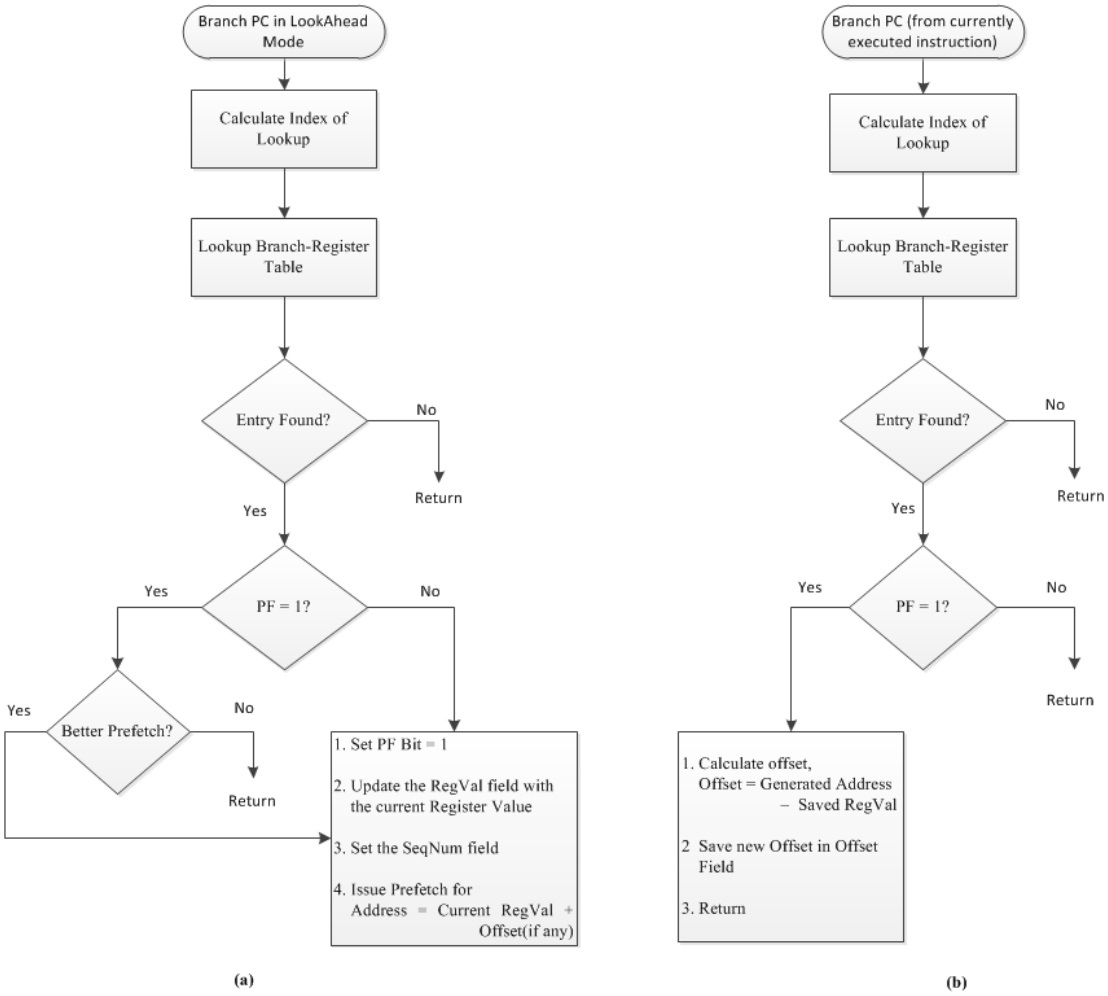


Figure IV.9. Flow Chart depicting (a) Process to use offset-field for prefetching; (b) Process to update the offset-field

The flow chart depicting the process followed to learn the offset values is shown in Figure IV.9 (b). As discussed before, the offset holds the difference between the actual address generated by memory instructions and the values of the corresponding

registers (used for those address computation) at a prior branch instruction. Thus, the key to learn offsets is to calculate them as instructions in the basic block get executed. Whenever a memory instruction executes in the main execution pipeline, it sends its generated address and its previous branch PC to the BrReg Table. The table gets looked up using the branch PC and in case the corresponding block is found in a prefetched state, it updates the corresponding offset values by computing the difference between the currently generated data address and the stored value in the RegVal field. After all the instructions in a basic block get executed, the PF bit is reset to 0, indicating that the required offset values were recorded for that entry and that the entry is ready to issue a fresh round of prefetch.

2. Loop-based Technique

This optimization was adopted to take advantage of loop-based behavior of applications. Many applications spend significant portion of their execution time executing loop-based codes. To efficiently and accurately prefetch for loops, our prefetching algorithm was modified to be able to identify loops using a hardware-only approach and generate prefetch addresses for the future iterations. The required modifications to the BrReg table entry are as shown in Figure IV.10:

Figure IV.10 Single Branch-Register Table entry (Loop implementation)

Each entry has been extended to contain 4 additional fields:

- Delta – This field holds the difference between the generated memory address values over consecutive execution instances of the same

instruction. It is analogous to the concept of stride, as used in traditional stride-based prefetching mechanisms [12], [20].

- Delta-Valid – This 1-bit field is used to find out if the instruction (to which the register value corresponds in the basic block) has been assigned a valid delta value or not. This bit will be set for those instructions which have been identified to be looping in some previous execution instances.
- Delta-is-Changing – This 1-bit field allows hardware identification of loops, as will be explained later in this chapter and aids in setting of the appropriate delta value.
- Loop-Counter – This field is used to monitor the iteration count of the loop in the lookahead mode. This allows accurate prefetching for data, to be used in a future iteration of the loop.

The basic operation remains the same as described with the previous implementations; however certain special measures are taken to ensure that the loop-based behavior is essentially captured and exploited in hardware. As our look-ahead scheme is capable of jumping across basic blocks in a single clock cycle, loop identification up to a certain nesting depth becomes fairly simple. For example, if there is a loop in the code given as follows:

```
Start: Load r1, 24(r2)
       Lda r2, 128, r2
       Cmpeq r2, r3, r1
Br1 : beq r1, start
```

Given that the path confidence is high, the look ahead procedure should yield the following sequence of branch addresses: - br1(Taken) → br1(Taken) → br1(Taken), the depth being determined by either when the confidence falls below a threshold or the maximum look ahead degree is reached. The loop-detection algorithm capitalizes on this idea that if during one complete look

ahead process, the same branch is visited more than once, it implies that a loop is most likely going to get witnessed. However, this technique implies that identification of loops with nesting depth greater than the maximum allowed look ahead degree is infeasible. Keeping this in mind, we describe the algorithm to deal with loop-based codes in the following paragraphs.

To be able to issue prefetches for a future iteration of any loop, three pieces of information are required: a) One is the Offset value, which captures the difference between the register values at a prior branch instruction and the actual generated address value. b) Second is the Delta value, which captures the difference between the register values over two consecutive iterations of the loop. c) Third is the loop-iteration count. If the above information is available, prefetch addresses for future iterations of the currently executing loop can be calculated at a branch-instruction as follows:

Prefetch Address = [Register Value] + Offset + (Loop-Counter * Delta)

Offset value calculation is relatively straightforward. It gets computed as memory instructions in a block get executed by computing the absolute difference between the generated address value and the value of the register, as saved in the RegVal field. Delta value calculation is slightly more involved. Delta value corresponds to the difference in the generated address values over consecutive iterations, and so, to estimate Delta, the values of the corresponding registers need to be monitored over consecutive iterations. This requires some changes to the algorithm used in previous implementations. In the only-offset case, as a basic block of instructions finished execution, the corresponding PF bit in the BrReg Table was reset to 0 to allow new prefetches to be issued for the basic block, at a future execution instance of the same. In this case, as a basic block ends, in addition to resetting the PF bit, the Loop_Counter value is also monitored. A value greater than 0 implies that this entry was visited more than

once during the current look ahead process and is hence, likely to be a part of a looping sequence. In this case, the Delta-is-changing field is set to 1 and the most recent value of the linked-registers is saved in the RegVal field. This step is done to allow setting of the delta value, the next time another dynamic instance of this basic block ends.

The flow chart depicting the process followed to generate prefetch addresses in the loop-mode is shown in Figure IV.11.

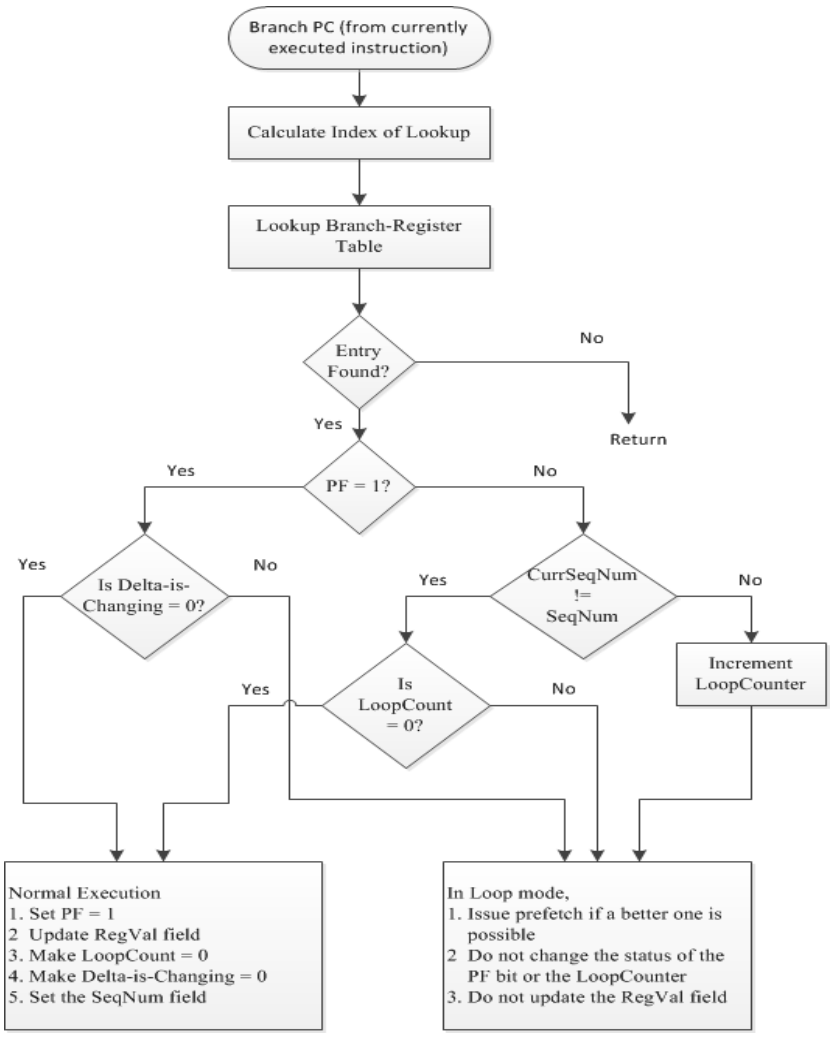


Figure IV.11 Flowchart describing the process to generate prefetch addresses in the loop-mode

When a prefetch is going to be initiated for a particular basic block (identified by the entry branch instruction), two different situations can arise depending upon the prefetched state of that block:

- The block is in a prefetched state (PF bit = 1). This situation may arise in the following two cases: a) if this block was looked ahead starting from an older branch instruction, b) if this block was visited sometime before during the current look ahead process itself (Note that this condition is pertinent to loop-handling).

  To distinguish between the above two cases, we make use of the seqNum field. As mentioned before, this field contains the sequence number of the branch instruction that had initiated the lookahead process and hence had led to the prefetch of the block in question. If a BrReg Table lookup request is generated for a branch instruction, whose corresponding entry is in a "Prefetched State", we compare the seqNum field saved in the entry with the SeqNum of the current look-ahead process.

  - In case they are equal, it implies that this basic block is being visited again during the same look-ahead cycle. This satisfies our condition for identification of loops. In this case, we do not update the value in the RegVal field to allow proper updates of the offset and delta fields (when this basic block instance ends). But so as not to lose opportunity for prefetch, we allow prefetches to be generated for this basic-block if a better prediction is available. We also increment the Loop Counter so that the next look-ahead into the same entry can prefetch for a different iteration. Note that in this mode, we do not prefetch an entire spatial region, but only the requested address.

- If the SeqNum fields do not match, it implies that this basic block was mostly looked ahead from an older branch instruction. Here, we allow prefetches to be issued in the event of availability of a different region prediction and we update the RegVal field and the SeqNum field to the latest values. Note that, in this case prefetches are issued for the entire spatial region as loop-behavior could not be established.

- The block is not prefetched (PF bit = 0). This state may arise if the block was never prefetched before or was prefetched and thereafter, cleared upon the completion of execution of the basic block. As we discussed before, if a loop was identified at the end of execution of some basic block, its PF bit would be reset to 0, but its Delta-is-Changing bit would be set to 1 (to ensure that upon completion of the basic block again, the delta values of the entry can be recorded). So, the required operation depends on the value of the Delta-is-Changing bit as follows:

  - Delta-is-Changing = 0: This implies that if at all the block was prefetched some time back, still no loop behavior was observed for it. This is the normal execution case. Hence, we can issue a prefetch for a spatial region, given by the register value plus the recorded offset (if any).
  - Delta-is-Changing = 1: In this case, to avoid prefetching opportunity, we issue prefetches only if we have some better prediction at hand. But we do not update the RegValue field as we need to record the Delta value with respect to the last iteration, when the basic block finishes execution this time.

IV.2.4 Prefetch-filtering Mechanisms

Aggressive prefetching mechanisms tend to bring in a lot of data into the cache, with an objective of reducing the number of cache misses. But if the prefetcher's accuracy is not high, the prefetcher might bring into the cache a lot of data that will not be needed by the processor before being evicted. Such data would still evict other potentially useful data from the cache and may deteriorate performance. Such a phenomenon is termed as cache pollution. Ineffective prefetches will also impact the bus bandwidth utilization, leading to further degradation in performance. So, it becomes important to control the number of useless prefetches and bring in only those data that have a higher likelihood of being used by the processor. This becomes even more important for systems which prefetch directly into the L1 cache, as its size is typically small and hence, it is not much tolerant of pollution.

As discussed before, our prefetcher tries to aggressively look ahead across basic blocks and exposes memory instructions in the same. But even after using the offset and loop-based enhancements discussed before, the prefetches issued for many basic blocks ahead may not be highly accurate. To not to lose opportunity, we therefore allow prefetches to be issued for already-prefetched basic blocks if better and more accurate predictions are available as program execution moves ahead. But the inaccurate prefetches that were identified in the Auxiliary pipeline (A.P) earlier may be detrimental to performance. To reduce the impact of such inaccurate and useless prefetches, we employ a few prefetch filtering techniques that scan the stream of prefetch requests sent out by the A.P and filter out the potentially useless prefetch requests. This section discusses in detail the filtering strategies that have been implemented in our current work. It is to note that not all of these techniques are a part of the final implementation, but are discussed here for completeness sake.

1. **Region-filtering FIFO Buffer:** As has been discussed, in the normal mode of execution, we prefetch an entire spatial region around a predicted candidate address. So, as a basic filtering strategy, we attempt to avoid prefetching for the same spatial regions in close succession. In order to achieve this, we employ a 3-entry FIFO buffer that sits in between the A.P and the prefetch queue and caches the 3 most recently prefetched spatial region addresses. When a request is issued in the A.P to prefetch a region of data, this structure is queried to check if that region has already been prefetched recently. In case the corresponding region is found in this buffer, the prefetch request gets discarded. Otherwise, the addresses are allowed to be queued up for prefetching. This filter is a part of our final implementation.

2. **Region Based Filter:** This filter determines the usefulness of predicting an entire region around a predicted data address. This is important because if a spatial region is such that only a few blocks in that region tend to get used, then prefetching the whole region upon a request would generate a lot of useless prefetches. Hence, to avoid this, we maintain a table of 3-bit saturating counters that gets indexed using a strong hash of the region address. The entry counters are incremented whenever a prefetch to a block in the corresponding region turns out to be useful or there is a demand miss to a block in that region and gets decremented in the event of a useless prefetch. Hence, at any time, a high counter value implies that the region incurs a lot of demand misses or most of the prefetches issued to this region tend to be useful and vice versa. Whenever a prefetch request for a region of data is issued, this table is queried using the hashed region address. If the corresponding counter value is higher than a pre-set threshold, then a region prefetch is initiated. Else the prefetch request gets discarded.

By employing this technique, the total number of useless prefetches that are issued gets significantly reduced and performance gets improved due to reduced pollution. However, this technique filters out prefetch requests for even those regions, which have sparse but very predictable access patterns. This is because this approach cannot distinguish between the different execution phase and also, the different instructions accessing that region.

3. **Path-trace Based Prefetch Filter:** To overcome the issues associated with the above approach, we propose another filtering technique that takes into account the program phase and the context of the prefetch to differentiate between the useful and the useless prefetches. Our look-ahead mechanism allows us to look-ahead from a branch instruction to as many basic blocks ahead as possible, till the path confidence falls below a certain threshold. But the addresses generated so many basic blocks ahead many not always be accurate. So, we employ a path-based index to assess the likelihood of correctness of the generated prefetches. But, again the basic blocks may contain a varied set of memory instructions which operate on different data structures and generate different access patterns and hence, it would be incorrect to assume that all such instructions would exhibit similar behavior in the lookahead process. While many of these instructions may not be predictable at very old branches, many others might be. So, it is accurate to assume that none of the instructions in a basic block would have a predictable pattern from many basic blocks before. Hence, in addition to a path-based trace, we also take into account the individual instruction (captured in the source register index) behavior in order to categorize prefetches into useful or useless categories.

   This filter is arranged as a prediction table, where the index is obtained by hashing the path-based trace and the register index (used as a source register in the corresponding memory instruction. Our prediction table consists of rows of

3-bit counters where, each counter corresponds to a block in the spatial region. Considering a 512 Byte spatial region and a 64 byte cache block size, in the current implementation, each row consists of eight 3-bit saturating counters. A high counter value implies that, when this sequence of branches had made a prefetch prediction last time for this memory instruction (represented by the hashed register index), the corresponding prefetches had turned out to be useful or vice versa. Thus, lower the counter value, higher is the probability that the block would not be useful to the processor. We define a ***"critical block offset"*** as the offset of the cache block (in the spatial region) that was predicted in the A.P using the register and the offset values. Patterns of useful or useless prefetches are learned with respect to the critical block offset. We also employ rotation about the critical block offset because it will take into the consideration the variable alignment of the data structures in a spatial region.

This scheme requires keeping the hash of the path and the register index that initiated the prefetch, together with each block in the cache tag array. So, whenever there is a demand request for a prefetched line, this index can be used to lookup the filter table and increment the saturating counter corresponding to that cache block. Similarly, in the event of a useless prefetch, the corresponding counter value of the entry can be decremented.

Finally, these learned patterns are used to guide prefetching as follows: - Whenever a prefetch request is issued from the A.P, the filter table is looked up using the hash of the path trace and the register index. Then, the selected row of counters gets translated into a useful prefetch-vector (a string of 1's and 0's to differentiate the useful and useless blocks in a spatial region), by comparing the individual counter values against a pre-determined threshold. This vector is then rotated about the critical block offset and is used to generate the addresses of the blocks that should be prefetched in that spatial region.

4. **Prefetch Queue Based Filter:** As our prefetcher tries to lookahead and prefetch for future basic blocks, a situation may arise when the prefetch candidates remain queued in the prefetch queue while the main pipeline starts executing the corresponding basic blocks. This may happen fairly often as in the current system, demand requests are given higher preference over prefetched requests and hence, prefetched requests get issued to the cache only when the cache tag ports are unused and the bus is idle. In such cases, if these prefetch requests are allowed to remain queued and are issued later when the opportunity arises, it may lead to issuing of significant number of unnecessary prefetches. It might also delay the issue of prefetches that are predicted for more recent basic blocks. To avoid such a situation, we employ a prefetch-queue cleanup mechanism in which we maintain certain state information per prefetch request entry to help remove those prefetch requests that are queued for older basic blocks and make room for new ones. We maintain the following information per prefetch request in the queue: - a) a 5-bit field to hold the last five bits of the program counter (PC) of the immediately preceding branch instruction, as an indicator of the basic block for which that prefetch was issued; b) a 1-bit field to indicate if the prefetch address was generated as a result of the branch-lookahead process. The modified prefetch queue is shown in Figure IV.12.

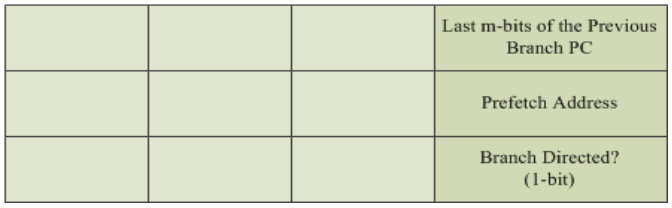


Figure IV.12 Modified Prefetch-Queue

The filtering process is explained in Figure IV.12. It is assumed in this example that each spatial region consists of two cache blocks. Figure IV.13 (a) depicts the process followed when Br 1 gets decoded: - A lookahead process is initiated that

issues prefetches for the immediately following basic block BB1 (see A1 and A2 addresses are queued up) and BB2, the basic block following Br 2 (see B1 and B2 addresses are queued up). The corresponding PC field gets filled with last few bits of Br 1 and Br 2 respectively. As execution continues and basic block BB1 retires, the address blocks that were identified for prefetch for BB 1, but are still queued in the prefetch queue get filtered out. This process is depicted in part (b) of the figure. This filter is a part of our final implementation.

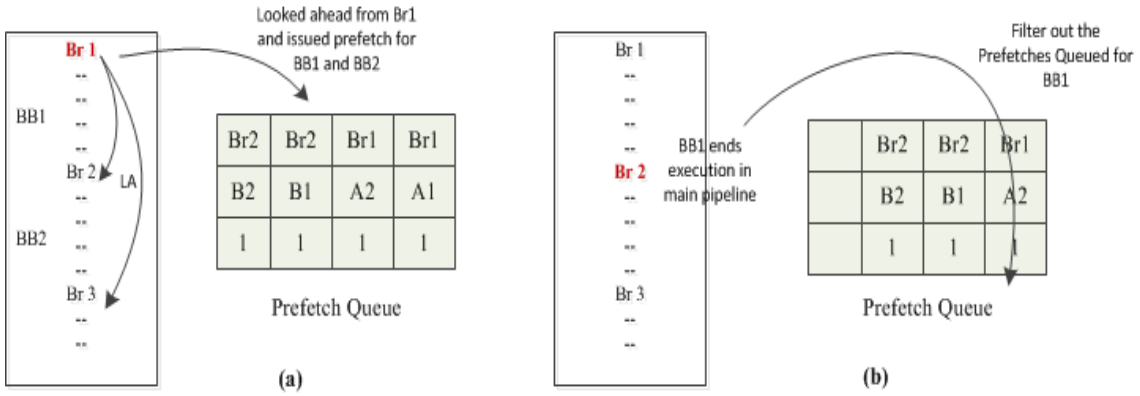


Figure IV.13 Example of the working of Prefetch-Queue based filtering

IV.3 Working Example

This section describes a detailed example of how the different structures described in the previous sections, work together to realize accurate and timely data prefetching. The working will be explained with the respect to program sequence given in Figure IV.3 (a). This program sequence consists of 4 basic-blocks of instructions (numbered 1 through 4). We shall start by discussing the procedure adopted to train the predictor structures. Thereafter, we shall discuss the procedure to make use of the trained state to issue data prefetches.

The learning phase, in which the table entries get filled works as described below: Assume that predictor tables are not trained at the start of execution of this program

sequence. This state is depicted in Figure IV.14 (a). The process is explained with respect to the first basic block of the program sequence. Note that in this state, no branch-directed prefetching can be initiated. We employ commit-time updates of the predictor tables to avoid pollution due to speculative entries.

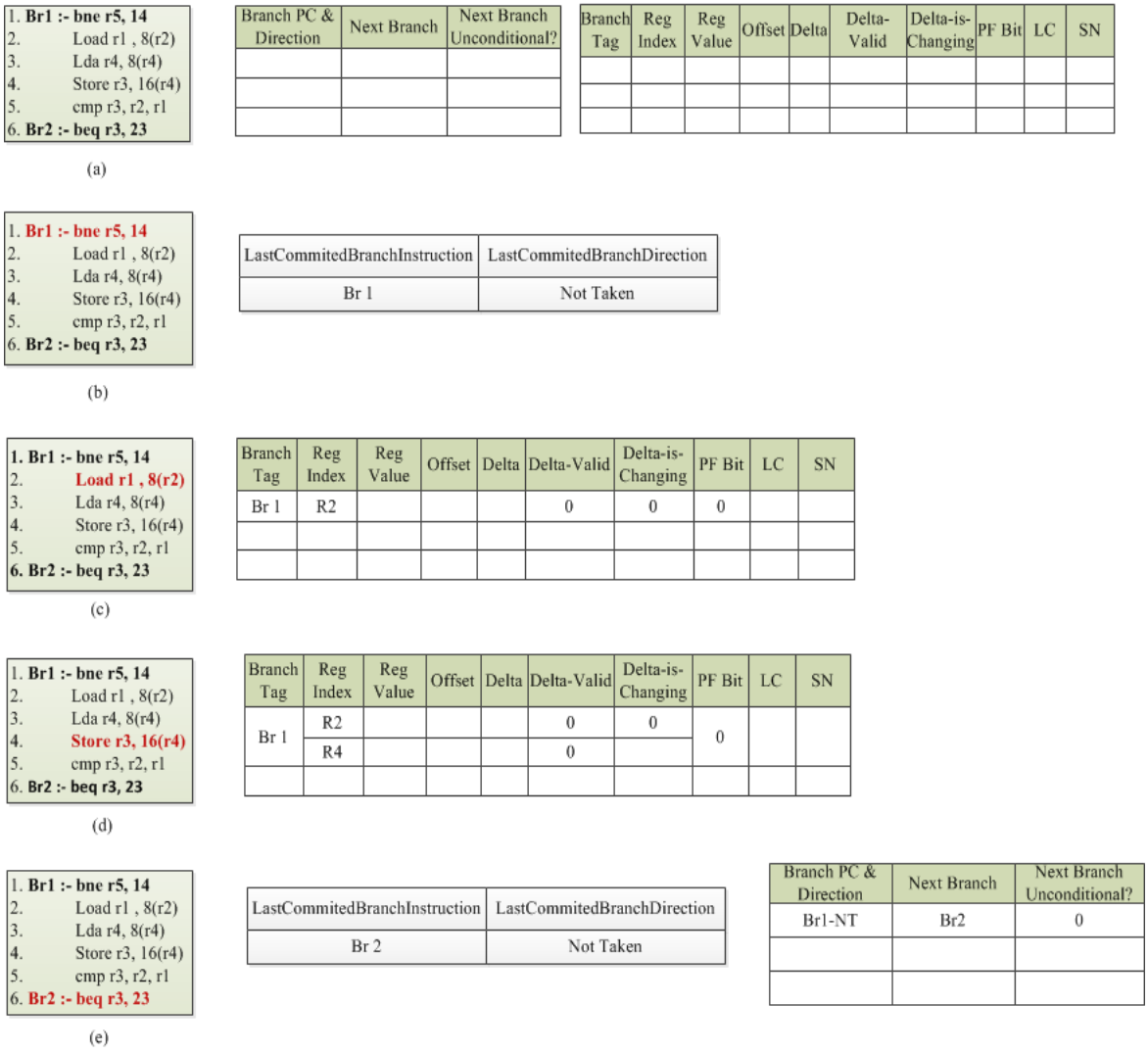


Figure IV.14 Working example showing update of Branch Trace Cache and creation of Branch-Register links

As the program execution starts and control reaches a stage when the instruction corresponding to SeqNum 1 is ready to retire, the LCBI register is loaded with the PC of

Br 1 and the LCBD register is loaded with Br 1's direction of execution (Not Taken here). This procedure is adopted to allow subsequent memory instructions to be linked to their preceding branch instruction and is shown in Figure IV.14 (b). As program execution proceeds further and SeqNum 2 (a memory instruction) retires, it gets linked to the branch Br 1, which led to the execution of this basic block and is held in the LCBI register. This link is shown in the BrReg Table in Figure IV.14 (c), where Br1 now gets associated with register R2 (the source register used for address computation in SeqNum 2). Similarly, when SeqNum 4 commits, it also gets linked to Br 1 in a similar manner and the BrReg table is updated to contain a link between Br 1 and register R4 (Figure IV.14 (c)). The above description explains how the branch-register links are created in the BrReg Table. Proceeding likewise, as SeqNum 6 (the next branch instruction in the program flow) commits, it finds that currently, the LCBI register holds the PC of Br 1 and LCBD register holds the last direction of execution of Br 1. At this stage, it can be inferred that if Br 1 executes in the direction given by the LCBD register, Br 2 would be the next branch to be encountered along that path (this argument is not applicable for branches with multiple target sites). This information is sufficient to re-create the execution path starting from Br 1, if it is encountered again in the future. This information is saved in the BrTc, as a link between Br 1 and Br 2 along the not-taken path of execution. This is shown in Figure IV.14 (d). Similarly, as the basic blocks 2, 3 and 4 complete execution, the rest of the entries of BrReg Table and the BrTc get updated. The final state of the predictor tables after the program sequence gets executed is shown in Figure IV.15.

Branch PC & Direction	Next Branch	Next Branch Unconditional?
Br 1 - NT	Br 2	0
Br 1 - T	Br 4	1
Br 2 - NT	Br 3	0
Br 3 - NT	Br 4	1
Br 3 - T	Br 2	0
Br 4 - T	Br 6	0

Branch Tag	Reg Index	Reg Value	Offset	Delta	Delta-Valid	Delta-is-Changing	PF Bit	LC	SN
Br 1	R2					0	0		
	R4								
Br 2	R1					0	0		
Br 3	R29					0	0		
Br 4	R5					0	0		

Figure IV.15 Trained State of prediction tables, corresponding to the program sequence given in Figure IV.3 (a)

Next, we shall discuss how the trained state is used to generate prefetch candidates. But before moving on, it is important to note that the learning of the structures and the usage of the learned state to issue prefetches happens continuously over the program run, though to initiate prefetching, at least some part of the structures must have been learned.

Assume that the program execution continues and at a later cycle, instruction Br 1 is encountered again. As Br 1 gets decoded, it gets fed into the Decoded Branch Buffer, from where it is fetched by the Auxiliary Pipeline (A.P). Thereafter, the A.P initiates the lookahead procedure, starting from Br 1. Figure IV.16 describes the different steps involved in the lookahead process in a typical case and the following paragraph explains the steps involved as well.

Cycle	BrReg Table Lookup Index	Branch Trace Cache Lookup Index	Branch Trace Cache Lookup Result	BrReg Table Lookup Result	Prefetch Address Generation Stage Result
C	-	Br 1 – NT	Direction = NT, High Confidence, Next Branch = Br 2	-	-
C+1	Br 1	Br 2 – NT	Direction = NT, High Confidence, Next Branch = Br 3	Reg Val[R2] Reg Val[R4]	-
C+2	Br 2	Br 3 – NT	Direction = NT, High Confidence, Next Branch = Br 4	Reg Val[R1]	Region Addresses [R2] and [R4] queued
C+3	Br 3	Br 4 – T	Direction = T, Low Confidence	Reg Val[R29]	Region Address [R1] queued
C+4					Region Address [R29] queued

Figure IV.16 Working example showing the prefetch address generation process

At cycle C, the BrTc is looked up using the Branch Br1 and its predicted direction of execution. In case the gross path confidence is above a threshold, the lookahead is allowed to proceed. Thus at the end of the cycle C, the BrTc yields us the next branch likely to occur along the path, i.e., Br 2. In cycle C+1, the BrReg Table is looked up

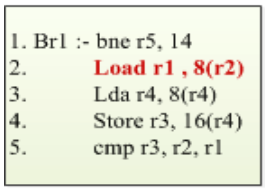
using the tag Br 1, in order to expose the memory instructions in basic block following Br 1. A lookup of the BrRegTable using Br 1 tag returns that R1 & R4 are linked to it and that this block has not been prefetched yet (given by the PF Bit being 0). At this stage, the basic block (BB 1) is marked as prefetched and the values of R1 and R4 (obtained by looking up the separate register file) are stored in the RegVal field of the entry and the corresponding PF bit is set to 1. In the meantime, Br 2 continues the look-ahead process by invoking the Branch predictor and the BrTc and Branch Br3 along the Not-taken path is identified with high confidence. In the next cycle, the data addresses generated for basic block BB1 are sent to the prefetcher to be queued after passing through the prefetch filter. At the same time, a lookup of the BrRegTable using Br 2 tag is initiated, that exposes the Register Index R1 and also yields that BB2 block has not been prefetched yet (given by the PF Bit being 0). At this stage, the basic block (BB 2) is marked as prefetched and the value of Register R1 (obtained by looking up the separate register file) is stored in the RegVal field of the entry. In the meantime, in the first pipeline stage, Br 3 looks ahead and predicts that Br 4 will be the next branch along its predicted direction of execution. This process keeps continuing till the predicted path's gross confidence falls below a certain pre-set threshold or else, the maximum allowable degree of look-ahead is reached. Like in this example, in cycle C + 4, Br 4 is predicted with a confidence value that makes the path confidence fall below the threshold. This terminates the look-ahead process. The final state of the tables after the lookahead process and prefetching is complete for this program sequence is shown in Figure IV.17.

Branch PC & Direction	Next Branch	Next Branch Unconditional?
Br 1 - NT	Br 2	0
Br 1 - T	Br 4	1
Br 2 - NT	Br 3	0
Br 3 - NT	Br 4	1
Br 3 - T	Br 2	0
Br 4 - T	Br 6	0

Branch Tag	Reg Index	Reg Value	Offset	Delta	Delta-Valid	Delta-is-Changing	PF Bit	LC	SN
Br 1	R2	[R2]	-	-	0	0	1	0	1
	R4	[R4]	-	-	0				
Br 2	R1	[R1]	-	-	0	0	1	0	1
Br 3	R29	[R29]	-	-	0	0	1	0	1
Br 4	R5	-							

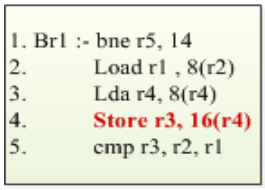
Figure IV.17 Snapshot of the predictor tables at the end of a lookahead phase

As the instructions in the basic block start executing, they update the offset and delta fields of the prefetched entries in the BrReg table. A typical example is shown schematically in Figure IV.18.



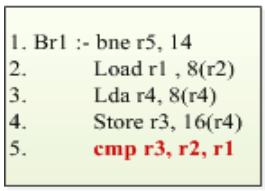
(a)

Branch Tag	Reg Index	Reg Value	Offset	Delta	Delta-Valid	Delta-is-Changing	PF Bit	LC	SN
Br 1	R2	[R2]	8	-	0	0	1	0	1
	R4	[R4]	-	-	0				
Br 2	R1	[R1]	-	-	0	0	1	0	1
Br 3	R29	[R29]	-	-	0	0	1	0	1
Br 4	R5	-							



(b)

Branch Tag	Reg Index	Reg Value	Offset	Delta	Delta-Valid	Delta-is-Changing	PF Bit	LC	SN
Br 1	R2	[R2]	8	-	0	0	1	0	1
	R4	[R4]	16	-	0				
Br 2	R1	[R1]	-	-	0	0	1	0	1
Br 3	R29	[R29]	-	-	0	0	1	0	1
Br 4	R5	-							



(c)

Branch Tag	Reg Index	Reg Value	Offset	Delta	Delta-Valid	Delta-is-Changing	PF Bit	LC	SN
Br 1	R2	[R2]	8	-	0	0	0	0	0
	R4	[R4]	16	-	0				
Br 2	R1	[R1]	-	-	0	0	1	0	1
Br 3	R29	[R29]	-	-	0	0	1	0	1
Br 4	R5	-							

Figure IV.18 Working example showing Branch-Register Table learning process

After Br 1 gets decoded (which invokes the lookahead mechanism in the Auxiliary pipeline), execution continues normally in the main execution pipeline. The results of the actual execution are used to update the values of ***"Offset"*** and ***"Delta"***. When SeqNum 2 decodes, it would again get linked to its prior branch instruction (Br 1). This process is achieved by maintaining another single Register Entry called the LastDecodedBranchInstruction (LDBI) which caches the latest decoded branch instruction PC. Moving on, when SeqNum 2 gets into its execution stage and generates

its data address, the BrReg Table is looked up using Br 1 index and the Offset field of the corresponding entry (R2) gets updated to the difference between the currently generated address and the value stored in the RegVal field. This is depicted in Figure IV.18 (a). As execution continues, and the SeqNum 4 calculates its effective address, it also looks up the BrReg Table to update the offset with respect to its source register, Register R4 (see Figure IV.18 (b)). Finally, as the basic block ends, i.e., all the instructions in the basic block (BB1) finish execution, the entry in BrRegTable is marked un-prefetched (i.e., PF bit is reset to 0). This state is shown in Figure IV.18 (c). At this stage, the offset values with respect to the basic block BB1 are learned.

Cycle	BrReg Table Lookup Index	Branch Trace Cache Lookup Index	Branch Trace Cache Lookup Result	BrReg Table Lookup Result	Prefetch Address Generation Stage Result
C + 5	-	Br 2 – NT	Direction = NT, High Confidence, Next Branch = Br 3	-	-
C+6	Br 2	Br 3 – NT	Direction = NT, High Confidence, Next Branch = Br 4	Prefetch only if a better prediction is available. Reg Val [R1]	-
C+7	Br 3	Br 4 – T	Direction = T, High Confidence, Next Branch = Br 6	Prefetch only if a better prediction is available.	No Prefetch
C+8	Br 4	Br 4 – T	Direction = T, Low Confidence. Look-ahead Terminates	Reg Val[R5]	No Prefetch
C+9					Region Address [R5] queued

Figure IV.19 Working example showing steps followed when prefetch is issued for an already-prefetched block

In our approach, lookahead process is initiated at every possible branch instruction at their decode stage. This means that a situation can arise when the lookahead process starting from one branch instruction would prefetch for a certain number of subsequently following basic blocks during its lookahead process and the next decoded branch instruction also attempts to prefetch for the same basic blocks in its next lookahead

process. Certain steps are taken in our technique to avoid prefetching for the same basic blocks in close succession. We will now discuss the steps taken when prefetching is attempted for a block that is already in the “prefetched” state. This process is shown in Figure IV.19.

Assume that all the instructions in basic block 1 have been decoded and now, Br 2 instruction (corresponding to SeqNum 6) enters its decoding stage. Note that since the Basic Blocks 2, 3 and 4 were prefetched because of the look ahead initiated by Br 1 and have not been executed yet since then. This state is marked by the corresponding PF bits being set at 1. As Br 2 gets decoded, it also gets pushed into the DBB, from where it is taken up by the Auxiliary pipeline and a fresh lookahead cycle is initiated. Referring to the table, in cycle C+5, Br 2 looks up the BrTc and the branch predictor and at the end of this cycle, we get the next branch (Br 3) along the predicted direction (if the predicted confidence is high). In the next cycle, the BrReg Table is looked up using the branch Br 2. But unlike the previous case, since the corresponding entry is already in its prefetched state, new set of prefetches are allowed to be issued only if a better prediction is available (i.e., if the value of Reg 1 falls into a different region than the previous prediction). However in the absence of a better prediction, the look-ahead process is allowed to continue, but no prefetches are generated for this basic block. In the same cycle, Br 3 invokes the branch predictor and looks up the Trace cache structure to determine that Br 4 is most likely, the next branch to be encountered along the path. In the next cycle, again a prefetch would get issued for the basic block following Br 2 only if a better prediction is available. It is however note that owing to better confidence estimates, if lookahead process can extend beyond branch Br 4, then prefetches are allowed to be issued for the corresponding basic blocks, as described earlier.

CHAPTER V

EVALUATION

In this chapter, we first discuss our simulation methodology in Subsection V.1. Then, we evaluate the effectiveness of the branch-directed prefetcher by comparing its performance benefits and effectiveness against other state-of-the-art prefetchers in Subsection V.2. Finally, we provide an estimate of the hardware cost of the proposed prefetcher in Subsection V.3.

V.1 Methodology

We evaluate our prefetcher in a simulation environment based on the M5 Simulator [25]. M5 is an open-source simulator platform developed by researchers at the University of Michigan. The simulator is used to model a 1-wide, 5-stage inorder pipeline. It is to note that this reference configuration is quite conservative in the light of the performance benefits that can be gained by using any prefetching technique and a more aggressively pipelined configuration is likely to demonstrate greater benefits. All non-memory instructions are assumed to be executed in one-cycle. The assumed memory model consists of a 2-level cache hierarchy with a 64KB 4-way set-associative L1 ICache & DCache and a 2MB 16-way set-associative L2 Cache. Under our assumed model, L2 cache hits are serviced in 16 ns and memory accesses are serviced in 60 ns. Table V.1 shows the important baseline architecture parameters.

We run 18 benchmarks from the SPEC CPU2006 benchmark suite, compiled for the ALPHA ISA. The reference input set is used for each benchmark. The results presented in this thesis were generated by running each benchmark for the first 1.5 billion committed instructions. We classified the benchmarks into two categories (Prefetch-sensitive and otherwise) based on if they showed at least 2% performance benefit with a

perfect cache.

We modified the simulator to include the branch-directed prefetcher-specific structures. The simulator was then used for detailed cycle-level processor simulation.

Table V.1 Target microarchitecture parameters

Simulator	M5 Simulator, ALPHA ISA, System Emulation Mode
Architecture	5-stage Inorder Pipeline, 1-wide, 2 GHz Frequency
Branch Predictor	Tournament Predictor
BTB	4096 entries
Register File	32 Integer Registers, 32 Floating-point Registers
ICache / DCache	64KB, 4-way set-associative cache, 64 Byte Line size, 1 ns access latency, 10 MSHRs, 3 Cache Ports
L2Cache	2MB, 16-way set associative, 64 byte line size, 16 ns access latency, 20 MSHRs, 1 port
Memory	60 ns access Latency

In our current work, we compare the performance implications of employing several state-of-the-art prefetchers. We evaluate the benefits of employing a Next Line 1 (NL1) prefetcher (one that prefetches the successively following cache block following a cache miss). We also analyze the performance impact of a stride-based prefetcher. However, we have omitted its results in the discussion because the results were generally worse, except in a few benchmarks that exhibit regular-strided access patterns. We also test a system incorporating the Spatial Memory Streaming (SMS) – based prefetcher. As implemented in the most recent proposal of SMS for SPEC CPU2006 benchmarks [6], we consider SMS with 512 Byte spatial region size, a 64-entry accumulation table and a 2K-entry Pattern history table. Finally we estimate the performance impact of the proposed branch-directed prefetcher, when used alone (called Branch-Directed) and in conjunction with the NL1 scheme (called Branch-Directed+NL1, and as described in Subsection IV.1). The results of our evaluation are presented in the next subsection.

V.2 Results and Analysis

V.2.1 Impact on IPC

The first set of experiments demonstrates the impact of incorporating the proposed prefetcher (both the branch-directed and the branch-directed+NL1 configurations) on the system performance (IPC) as compared to a baseline (no-prefetching) system. In this experiment, we also compare the performance improvements gained by employing a next-line prefetcher, a Spatial Memory Streaming (SMS) – based prefetcher over the baseline (no-prefetching) system. The branch-directed system considered in this experiment, is based on the loop-based BrReg Table implementation (discussed in Subsection IV.3.3) and employs the Region-Filtering FIFO Buffer, the Region-Based Filter and the Prefetch-Queue Based filter (as discussed in Subsection IV.4) for filtering purposes. The results of our evaluation are presented in Figure V.1.

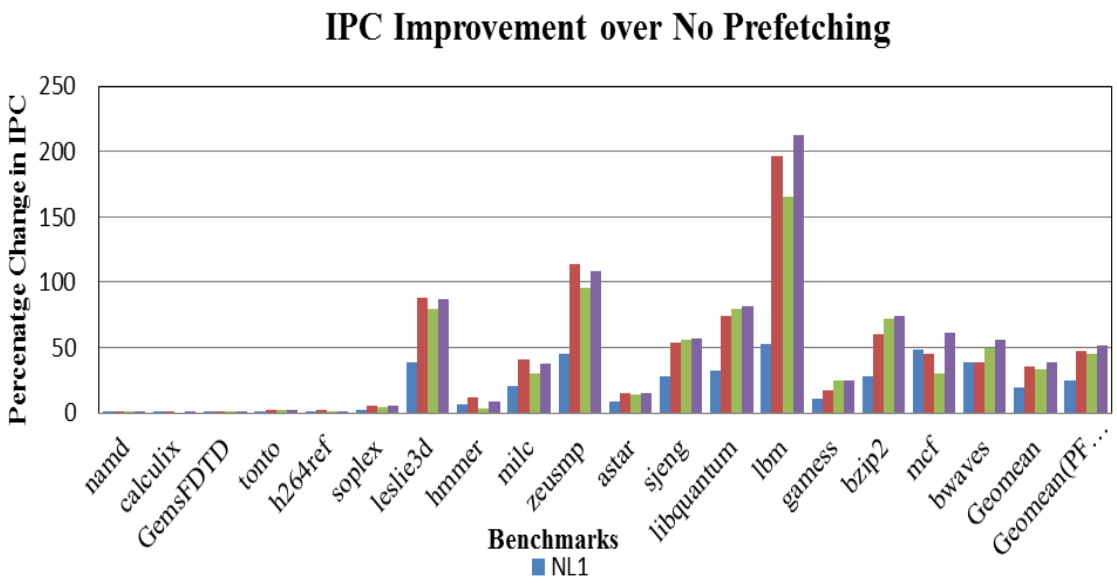


Figure V.1 IPC improvement over Baseline (No-prefetching)

As shown in the figure, there are four bars for each benchmark. The leftmost bar corresponds to the performance improvement when the Next-Line 1 (NL1) prefetcher is

employed alone over the baseline implementation. The second bar from the left corresponds to the performance gain by employing the SMS prefetcher. The third bar from the left corresponds to the performance of our branch-directed prefetcher alone, as compared to the baseline no-prefetching system. The rightmost bar corresponds to our prefetcher in conjunction with the NL1 scheme. The results show that the NL1 and the SMS prefetcher alone provide a performance benefit of 19.1% and 35.87% over the baseline system respectively. While, the branch-directed prefetcher, without and with the next-line improvisation provides a mean speedup of 33.63% and 38.789% over the baseline system respectively.

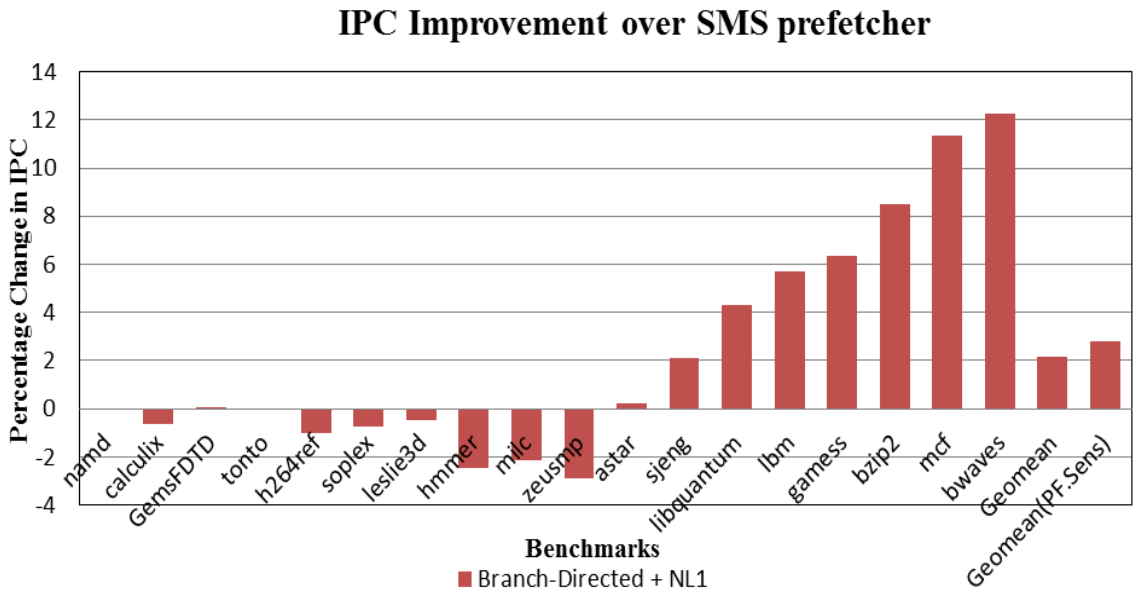


Figure V.2 IPC improvement over SMS prefetcher

Figure V.2 shows the performance impact of the branch-directed configurations as compared to the SMS-based one. From the figure, it can be observed that while the branch directed prefetcher degrades the performance by 1.645% (2%) when used alone, in conjunction with NL-1 prefetcher the performance improves by 2.148% (2.82%), over the SMS prefetcher, averaged across all 18 SPEC2006 benchmarks (only across the prefetch-sensitive benchmarks).

These results show that the branch-directed scheme can alone deliver almost the same performance benefits as a SMS-based prefetcher implementation. And, in conjunction with the NL1 scheme, it performs better than the SMS implementation. This implies that hybrid branch-directed+NL1 scheme is very effective at reducing cache misses. The branch-directed scheme can take advantage of spatial locality in an application as long as it can predict the region of operation accurately. It can also prefetch accurately for irregular and isolated data accesses. Additionally, the NL1 scheme provides benefit by exploiting spatial locality in the event of cache misses (which may occur if a branch-directed prefetch was either not issued in a timely manner or was not accurate enough or no prefetch was issued in the first place due to insufficient training of the structures).

Also, the SMS prefetcher predicts future memory accesses based on current memory misses and hence, cannot predict the first misses in a spatial region. For those applications that exhibit less dense spatial patterns, such misses also form a significant fraction of all misses and hence, the performance improvements gained by incorporating SMS is minimized in such cases. On the other hand, by decoupling prefetch decisions from the cache miss events, our prefetcher can accurately anticipate future misses and prefetch for them.

V.2.2 Prefetch Effectiveness

Another set of experiments is conducted to demonstrate the effectiveness of the prefetches issued by the branch-directed prefetcher as compared to the SMS prefetcher. The results of this experiment are shown in Figure V.3.

In this experiment, the effectiveness of the prefetcher is estimated by categorizing the total prefetches issued into useful (demand request for these data is received before their eviction from the cache), useless (the data gets evicted without receiving any demand hits) and untimely (the demand request gets issued while the data is en-route from the

lower levels of memory to the L1 cache) prefetches, normalized against the total number of prefetches issued by the SMS configuration. It can be observed from the graphs that for many benchmarks (like games, bzip2, mcf), the branch-directed prefetches are more accurate than those issued by the SMS prefetcher. However on benchmarks like leslie3d, SMS prefetcher is very accurate, and many prefetches issued by the branch-directed prefetcher are either useless or untimely. Thus, the SMS prefetcher performs better over the branch-directed prefetcher for such benchmarks. In others like milc, the branch-directed prefetcher prefetches a lot of prefetches (mostly useless) and hence, degrades performance. Interestingly, branch-directed scheme prefetches significant number of useless prefetches for the bwaves benchmark. However, this benchmark is tolerant of cache pollution and hence, it benefits from the increased number of prefetch hits.

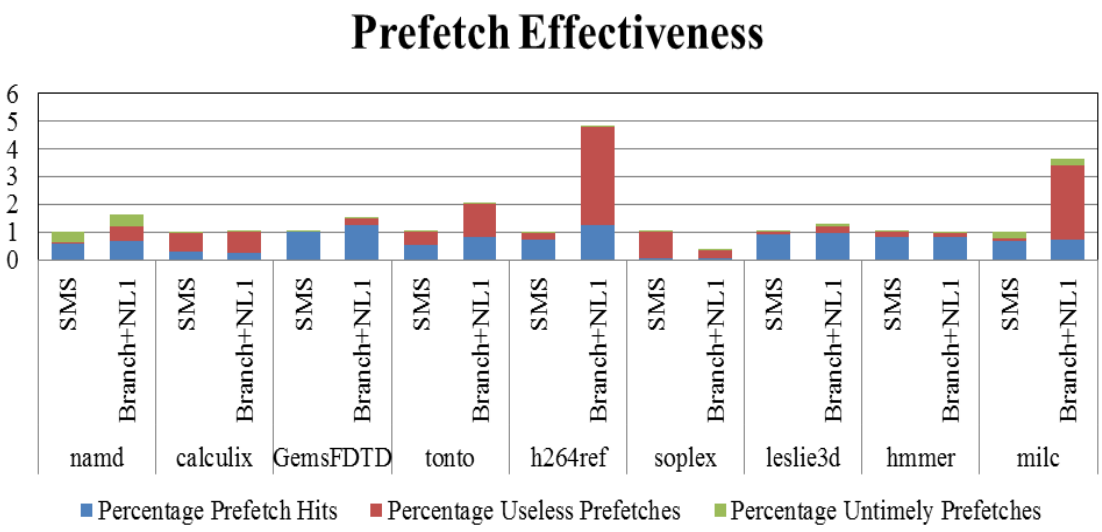


Figure V.3 Effectiveness of issued prefetches

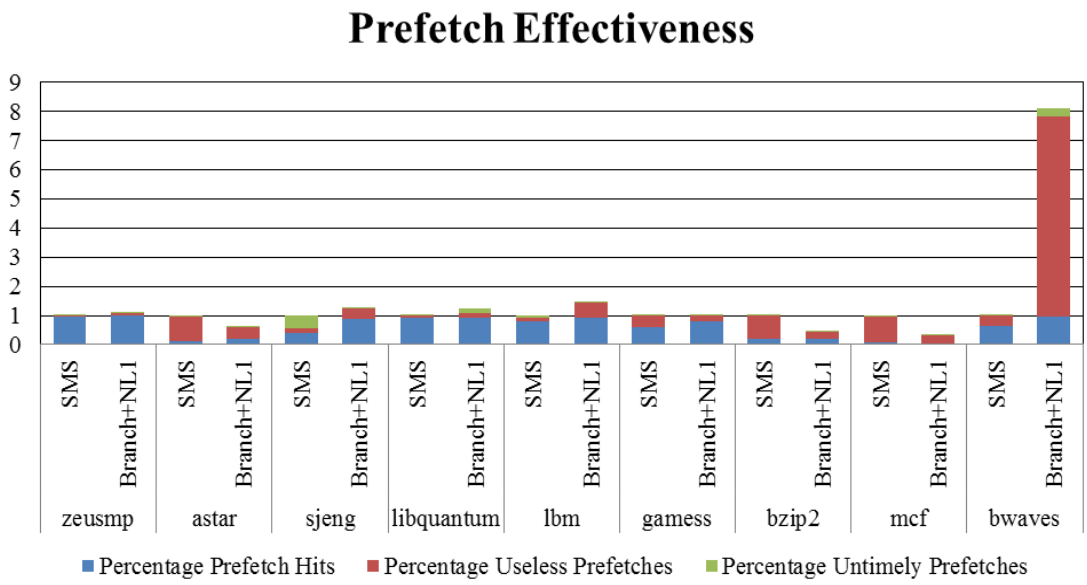


Figure V.3 Continued.

V.2.3 Bus Traffic

A third set of experiments is conducted to estimate the effect of the generated prefetches on the L1-L2 bus traffic. This effect can be approximated by monitoring the increase in the number of L2 accesses normalized against the baseline (no-prefetching) configuration. The result of this experiment is presented in Figure V.4. There are 4 bars for each benchmark, each bar depicting the increase in number of L2 accesses by employing the corresponding prefetcher over the baseline. It can be observed from the graph that the increase in number of L2 accesses is approximately 29.26%, 67.674%, 79.591 and 84.544% after incorporating the NL1, SMS, branch-directed alone and in conjunction with NL1 respectively.

It is interesting to note that for mcf which is a bandwidth constrained application, SMS generates a large number of prefetches. Hence, it significantly worsens the performance over the branch-directed prefetcher. It is however important to note that the branch-

directed prefetcher is more aggressive in nature and hence, on an average generates 12% more L2 accesses as compared to an SMS based implementation.

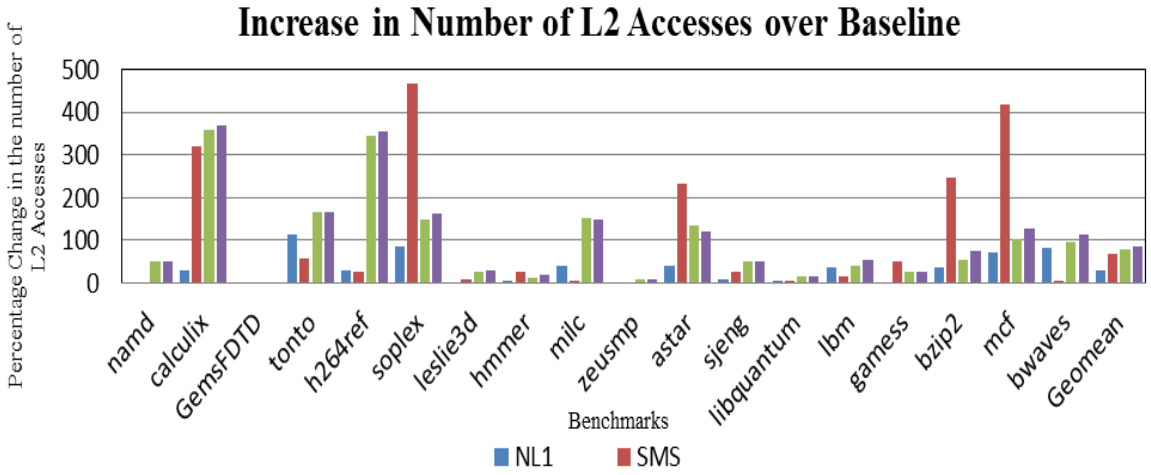


Figure V.4 Increase in number of L2 Cache accesses

From the figure, we can also observe that the branch-NL1 system generates approximately the same number of L2 accesses as the branch-directed system alone. This is because the total number of demand misses occurring in the branch-NL1 system is much less than those occurring in either a NL1-based or a branch-based system alone. This implies that the NL1 prefetcher is triggered less often in the combined branch-NL1 system, which ensures that the total number of requests (demand and prefetch) issued for the L2 cache remains virtually the same.

V.2.4 Impact of Predictor Table Size

We also conducted a preliminary set of experiments to evaluate the impact of predictor table sizes on the overall system performance. To evaluate the impact of the Branch-Register table size, we varied the number of entries in the table and monitored its impact on IPC. We observed that there is no significant change in IPC beyond a table size of

128 entries. A similar experiment using the branch Trace Cache demonstrated that a table size of around 128 to 256 entries is sufficient to deliver most of the performance benefits. Even though these results are preliminary, they demonstrate that our prefetcher can efficiently capture the variability in program behavior with smaller table sizes as well. This can be explained by realizing that in our approach, we establish branch-based correlation to enable prefetching. Any typical program has more number of memory instructions than control instructions. So, given that a memory instruction-correlating prefetcher (like, stride-prefetcher etc.) can capture the essential information, needed for prefetching, in 256-512 entries, we should theoretically capture the same amount of information at significantly reduced table sizes.

V.2.5 Hybrid SMS and Branch-directed Prefetcher

We conducted another set of experiments to analyze the performance impact of a hybrid prefetcher combining the SMS and the branch-directed prefetcher (SMS+Branch). To realize this, we made the following changes to our original implementation: - a) Instead of the region-filter (discussed in Subsection IV.4.2), we employed the Path-Trace Based Prefetch Filter discussed in Subsection IV.4 b) Also, given that the SMS prefetcher can take advantage of spatial locality in loop-based codes, we employed the offset-based variation instead of the loop-based one. The results of this experiment are shown in Figure V.5. It can be seen that the hybrid prefetcher provides a benefit of 37.82% (50.515%) over the baseline configuration across all the 18 SPEC benchmarks (over the prefetch-sensitive benchmarks). It also achieves a 1.436% (1.814%) improvement over the SMS prefetcher. These results imply that by exploiting branch-based correlation and the basic-block fast-forwarding mechanism, the branch-directed prefetcher can prevent even those misses which are not predicted by the SMS prefetcher.

It is however, to note that this hybrid system performs worse than the hybrid branch-directed+NL1 scheme (discussed before). This is because of the following reasons: a) Firstly, the branch-directed prefetching configuration assumed in the two hybrid schemes are different. The offset-based BrReg table configuration assumed in this case is less effective in generating accurate prefetches than the loop-based scheme because it fails to handle the loop-based applications. b) Secondly, both SMS and the branch-directed prefetchers are aggressive in nature. Hence, the total number of prefetches issued by the hybrid SMS-Branch implementation is significantly more than either prefetcher alone. These additional prefetches cause cache pollution and also, impact the demand on the limited bus bandwidth. This in turn, reduces the performance benefits achieved by prefetching.

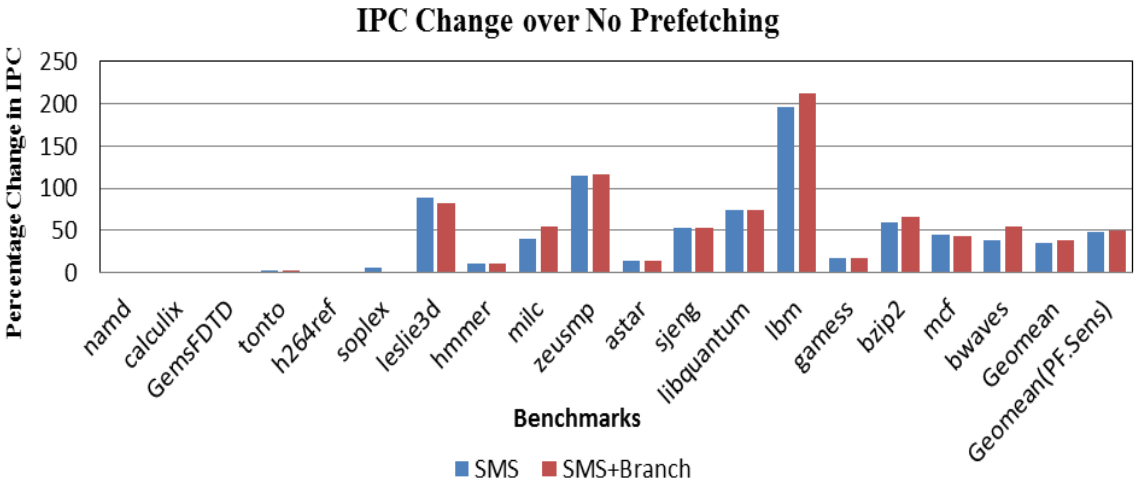


Figure V.5 Performance impact of hybrid SMS and Branch-Directed prefetcher

V.2.6 Inorder versus Out-of-Order: Impact on IPC

A final set of experiments was conducted to compare the performance impact of an inorder implementation (with prefetching support) over an out-of-order implementation (without prefetching support). As discussed previously, inorder cores are gaining more attention because of their low power and area requirements as compared to their superscalar counterparts. However, inorder processors have reduced single-threaded

performance because of their inability to work around cache misses. The motivation behind this experiment is to evaluate prefetching as a mechanism to improve the performance of inorder processors as compared to out-of-order (OOO) processors. In this experiment, the performance of an inorder processor equipped with a branch-directed prefetcher is compared against a 4-wide OOO implementation. The result of this experiment is presented in Figure V.6.

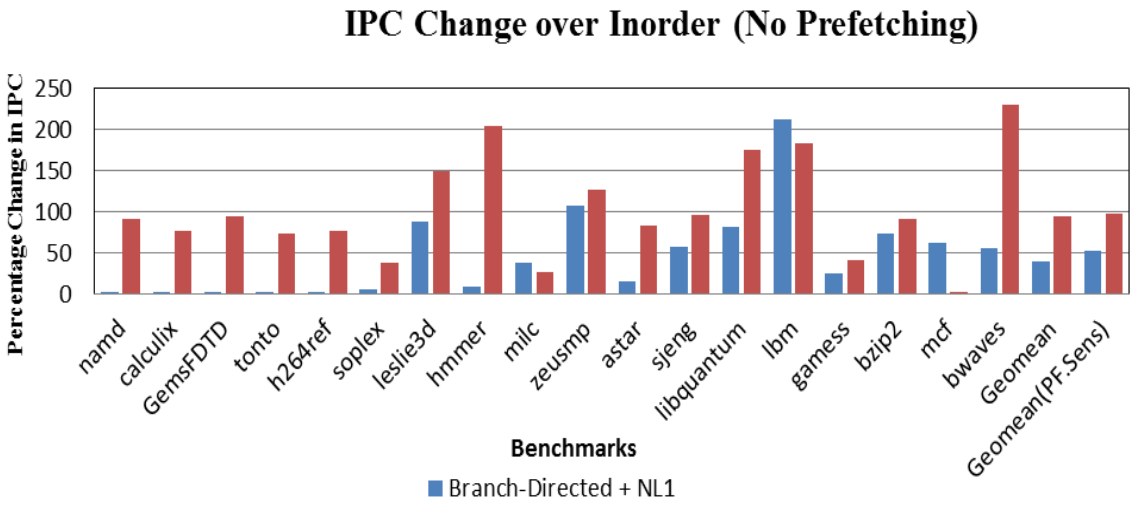


Figure V.6 Performance comparision of Inorder (with prefetching support) and out-of-order implementations

From the figure, it can be observed that while the prefetching-enabled inorder system provides a mean speedup of 39% over the baseline inorder system, the OOO system provides a 94% benefit over it. Thus, the inorder implementation provides roughly 42% of the benefit provided by the out-of-order implementation at a significantly reduced hardware overhead.

From all the experiments discussed in this chapter, it can be concluded that the branch-based prefetcher improves the performance of a system significantly. However, given its aggressive nature, there is further room for improvement if better prefetch-filtering

techniques are adopted. During the experiments, we also noticed that a number of prefetches issued by the branch-directed prefetcher are already residing in the cache. These additional prefetches consume power during the cache-tag lookup process and hence, certain measures should be adopted to limit them. This aspect can be explored in future work.

V.3 Hardware Cost

The additional hardware requirements of the branch-directed prefetcher can be summarized as follows:

- **Branch Trace Cache** – The current implementation of Branch Trace Cache has 256 entries. Each entry requires 66 bits: two 32-bit fields for the branch instruction PC, 1-bit for the direction of execution, 1-bit for the next-branch-is-unconditional field. Thus, the Branch Trace Cache requires 2KB of space.
- **Branch-register Table** – The current implementation of Branch-Register Table has 128 entries. Each entry requires 392 bits: 32 bits for the branch instruction PC, 350 bits for a maximum of five fields allowed for the Register-specific fields (5-bits of register index (RegIdx), 32-bits of register value (RegVal), 16-bits of Offset, 16-bits of Delta, 1-bit of Delta-Valid field), 1-bit for Delta-is-changing field, 1-bit for Prefetched (PF) field, 4-bits for the Loop Counter field and 4-bits for the Sequence Number field. Thus, the Branch Register Table requires a total storage of 6.125 KB.
- **Alternate Register File** – This unit has 32 8-byte entries. Thus, the Alternate Register File requires 256 bytes of storage.
- **Prefetch-region Filter** – This filter has 1024 entries. Each entry requires 3-bits. Thus, this structure requires an additional storage of 384 Bytes.

- **Path Confidence Estimator** – The confidence estimator has 2048 entries. Each entry has 8-bits (4-bits for the JRS Counter & 4-bits for the Up-Down Counter). Thus, this unit requires a total storage of 2 KB.
- **Modified Prefetch Queue** – Apart from the candidate prefetch address, each entry of the prefetch queue stores 5-bits from the previous branch instruction PC and a 1-bit field to distinguish the branch-directed prefetches from others. A 100-entry prefetch buffer is assumed in this implementation. So, the prefetch queue requires an additional storage of 75 bytes.
- **Others** – This prefetcher requires counters to estimate the misprediction rate of each confidence bucket dynamically. The current implementation requires a total of 74 counters for the 37 confidence buckets. Also, to enable filling of the Branch Trace cache entries, there is a need for a 32-bit LCBI register (to hold the last committed branch PC) and a 1-bit LCBD register (to hold the direction of execution of the last committed Branch PC). These structures together require approximately 300 bytes of additional storage.

Thus, the overall hardware cost of the branch-directed prefetcher is approximately 11.11 KB of storage, which is approximately 35% of the hardware overhead required by the SMS prefetcher.

CHAPTER VI

LESSONS LEARNED

This chapter discusses several variations to the base branch-directed prefetcher implementation that were attempted during this work. Subsection VI.1 describes an attempt to implement an efficient hybrid prefetching solution by using the set-dueling principles (as proposed in [26]). Subsection VI.2 discusses a modified Branch-Register Table implementation called the "Min-Max scheme" that is aimed at reducing the number of useless prefetches generated by the prefetcher. Subsection VI.3 discusses a modification to the base prefetcher that is directed at handling indirect branches. Finally, Subsection VI.4 discusses the impact of prefetching into the LRU position of a set so as to reduce the degree of cache pollution.

VI.1 Hybrid Prefetcher Implementation

We borrowed the set-dueling approach proposed by Qureshi et al., in an attempt to realize a hybrid prefetcher using the SMS and the branch-directed prefetcher [26]. The basic idea behind the approach is - Given that most of cache sets get used in a similar manner, a few sets could be dedicated to monitor the impact of different competing mechanisms on the performance. And finally based on such an analysis, the better-performance strategy could be used for the remaining cache sets. This idea was initially proposed in conjunction with L2 cache, which has a large number of sets and hence, it is plausible to dedicate a few sets to each strategy for monitoring purposes. This essentially avoids the need to maintain separate tag directories. However, in this thesis, we employ a prefetcher that prefetches directly into the L1 Data Cache. A typical L1 Cache has comparatively lesser number of sets than a L2 Cache. So, to exploit set-dueling in our case, we implemented separate tag arrays for each of the representative number of sets allocated for each competing strategy.

Essentially to realize a hybrid prefetcher using the Branch-Directed prefetcher and SMS, three sets of representative tag arrays were created: one each for analyzing the impact of spatial prefetching, only branch-based prefetching and the combined prefetching approach. Each category was composed of a few duplicated sets of the cache. The monitoring system is implemented in such a way, that the sets monitoring spatial-prefetcher's effect do not get affected by branch-based prefetches and so on. The program run was divided into fixed-length phases and the miss rate was observed in each of the representative sets during this phase. At the end of each phase, the observed miss rate of each representative set (as observed during the last program phase) was compared and the policy that yielded minimum miss rate was chosen to be the de-facto policy of all the actual sets of the cache.

Given that both SMS and branch-directed prefetchers are aggressive in nature, by naively combining the two approaches, a marginal degradation in performance was observed on a few benchmarks that are either bandwidth limited or not much tolerant of cache pollution. Hence by incorporating this approach of selecting the prefetching strategy depending on the observed miss rates in the representative categories, some performance benefits was recorded even for those benchmarks which had earlier showed degradation with the naïve-hybridization strategy. But the overall performance impact after adopting this technique was marginal, as compared to the increase in hardware overhead. So, this approach was not incorporated in our final implementation. However, this experiment demonstrated that set-dueling concepts can be used for implementing better and more accurate hybrid prefetching schemes.

VI.2 Modified Branch-Register Table Implementation (The Min-Max Scheme)

Min-max scheme was proposed as a modification to the Branch-register Table implementation in order to reduce the number of useless prefetches generated by the prefetcher. The previous proposal of Branch-Register Table required prefetching of an

entire spatial region around a predicted data address. This method tried to avoid this need by learning variable-sized regions around the predicted data address that are more likely to be useful. This mechanism is implemented as an enhancement to Offset-Only approach discussed in the Subsection IV.2.

In this strategy, instead of saving a single offset as suggested in Subsection IV.3, we save a range of offsets that captures better, variability of generated address values with respect to register values at prior branch locations. Because of saving such a range of offsets, prefetches are issued only for the blocks contained in this range, instead of the whole spatial region. The new BrReg Entry is given in Subsection VI.1:

Figure VI.1 Single Branch-register Table entry (Min-Max implementation)

The process adopted to link the branch instructions with memory instructions (more specifically, their source register indices) in subsequent basic blocks, is the same as discussed in Chapter IV.3. The difference lies in the mechanism adopted to learn the offset values and also to generate prefetch addresses. The offset-range is decided by observing the differences between the generated data addresses and the corresponding source register values at prior branch locations. In the current implementation, we again view memory as being composed of coarser spatial regions consisting of eight cache blocks each. But unlike the previous assumptions, the start address of the spatial region is assumed to be the block address containing the predicted data address. The minimum and maximum offsets basically denote the range by which actual data addresses falling into that spatial region, were different from prior register values in the past runs. For example, if a block A was predicted as a prefetch candidate for a future basic block during a lookahead process. When the basic block was actually executed, the generated block address was observed as A+2. At this point, the minimum and maximum offsets

both get set to a value of 2, corresponding to this basic block and register entry in the BrReg Table. If during another run of this basic block, the generated block address differed from the prefetched register value by 4, then only the maximum offset field gets updated to 4. This implies that a register value-to-actual address variability of 2 to 4 cache blocks was observed for this entry in the past runs. This information therefore, eliminates the need to prefetch a whole region around the predicted data address. Rather, only the cache blocks falling into the "RegValue + MinimumOffset and RegValue + MaximumOffset" range, get prefetched. Each register index in a branch-indexed entry is allowed to cache three such address ranges.

To understand the relative advantage of the min-max scheme over the single-offset scheme theoretically, consider the following code fragment:

```
Br 1 : beq r6, position1
       Load r3, 0(r2)
       Add r2, 64, r2
       Load r4, 0(r2)
       Add r2, 64, r2
       Load r4, 0(r2)
Br 2 : br position2
```

Let us assume that corresponding to this code fragment, an entry exists in the Branch-Register table that links Br 1 branch with Register R2. Then, as per the single-offset method discussed previously, we would save a single offset with respect to Register R2 and to allow prefetching for all the instances in the basic block, we would prefetch the region around the address given by the value of R2 at the branch-decoding instance. It can be observed that this method also allows issuing prefetches for all the instances of R2 in the basic block. But since in this case, a maximum of three different cache blocks will be touched during the execution of this basic block (assuming our baseline memory architecture), the remaining prefetches issued for that region tend to be useless.

However, following the min-max scheme, if a range of minimum and maximum offsets is maintained for each register index, the variability between the generated addresses

(using this source register) and the register value at a preceding lookahead instance can be captured with lesser number of prefetches. Like in this example, the offset that would be saved with respect to R2 would be 0(minimum) – 128(maximum). Thus, this approach has the potential to reduce the number of useless prefetches.
However after incorporating this modified system, we did not record significant performance benefit, except in some individual benchmarks. Hence, this mechanism requires further exploration in future.

VI.3 Indirect Branch Handling

In our current implementation of the Branch Trace-Cache, it is assumed that all control-instructions have a single possible target site along each direction of execution. But there exists a special class of control instructions (to support dynamically linked libraries, virtual function calls etc.), for which the direction of execution alone does not determine the subsequent basic block to be executed. This implies that for such branches, even if the branch predictor predicts the direction with high confidence, but because of the possibility of multiple target sites that can be dynamically invoked, the lookahead may move towards an incorrect path of execution. There are many possible alternatives to handle such branches. One possibility is to store all possible target sites starting from such branches in the Branch Trace Cache. Such optimization can be supported if the major classes of applications being serviced involve significant use of such branches. This alternative has not been explored in this thesis. Another alternative to identify and handle such branches in hardware is to incorporate another bit (called the “Stable Bit”), corresponding to each entry of the BrTc table. This bit indicates whether the “start branch PC” has always led to the “next branch PC” along the recorded direction in the past. The modified BrTc entry is shown in Figure VI.2:

Figure V1.2 Modified Branch Trace Cache entry to handle the indirect branches

It requires a small modification to the previously discussed update-procedure to train this modified scheme. This is discussed as follows: - As control instructions commit in program order, the corresponding Branch Trace Cache entries get filled out. However, during the update process, if a branch instruction is encountered that hits in the Branch Trace Cache, but its next-Branch field contains an entry that is different from the current "next-branch", then it can be inferred that this branch instruction has more than one possible target location along the same direction of execution. At this point, the stable-bit of the entry can be set to 1. Note that for the normal class of control-instructions which have a single target along each direction of execution, this bit remains at 0.

This modified scheme again introduces a minor modification to the lookahead process. In this case, a lookahead is allowed to proceed only if the corresponding branch entry hits in the trace cache (as before) and its stable bit is 0. This additional clause ensures that lookahead terminates at indirect branch locations, assuming that the subsequent path cannot be confidently established.

However, from the results of the preliminary experiments that we conducted using this modified implementation, we observed that this optimization did not improve performance much (although it is incorrect to generalize as the classes of applications used in this thesis do not need significant use of such branches). In fact for one case, it caused minor degradation in performance. The degradation was primarily observed because in many cases, though the target sites from the same branch PC are not unique; still these target sites have many memory reference instructions that use the same register indices as the alternate basic block. So, by limiting the lookahead beyond such

control instructions, some prefetching opportunity gets lost. Also, another possible cause of degradation can be that many such indirect jumps represent some kind of function calls. If this holds true, then even if the target site of the function call cannot be confidently established, the return path after the function call ends may be extremely predictable. Thus, even greater opportunity for prefetching would be lost by terminating the lookahead process prematurely.

VI.4 LRU Insertion Policy for Prefetched Blocks

As is known, Least Recently used (LRU) Policy is the standard cache replacement policy used in most of the modern microprocessors. Under this scheme, the victim chosen for replacement in a cache set is the block located at the LRU position of the set's LRU stack and the incoming block gets placed at the Most Recently used (MRU) position of the stack. The objective behind placing an incoming block in the MRU position is to give it an opportunity to be referenced by the CPU while it moves down the LRU stack. If prefetched data are also dealt in a similar fashion, then it is likely that more useful demand-hit data may be evicted out of the cache to make space for the prefetched data, which may be completely useless due to low accuracies of prefetchers. Since aggressive prefetchers tend to sacrifice accuracy for greater coverage, they bring in a lot of data into the cache that will never be used. This implies that most of the prefetched cache lines would simply go down from the MRU position to the LRU position of the stack, without receiving any demand cache hits. This further leads to the ineffective use of the caches. Hence, an attempt was made to assess the performance impact of a scheme that places all the prefetched data into the LRU position of the stack so as to reduce the cache pollution effects. In such a scheme, a prefetched block gets promoted to the MRU position only after receiving a demand request for the same.

From the preliminary set of experiments conducted in this direction, it was observed that this technique works well for those benchmarks, which suffered due to cache pollution

effects. However this technique reduces the benefit margins on those applications which showed significant benefits without this optimization. This behavior can be explained by the fact that the current lookahead mechanism prefetches for future basic blocks much ahead of their actual execution. In that case, if all the prefetched data gets placed into the LRU position, then there is a higher chance that prefetched data will get evicted before the appropriate demand request arrives. This will negate the advantages of prefetching. To avoid such a situation, prefetched data can be inserted into a different position in the LRU stack, other than the MRU and LRU. This aspect can be considered in future work.

CHAPTER VII

CONCLUSION AND FUTURE WORK

In this thesis, we proposed a data prefetcher that leverages the high prediction accuracies of current-generation branch predictors to accurately generate the future basic block trace that the program will follow and initiates data prefetching for memory instructions in those future basic blocks. We also demonstrate that there exists a strong correlation between the addresses generated by memory instructions and the values of the corresponding source registers at prior branch locations. In the proposed implementation, we exploit this correlation by establishing links between the branch instructions and register indices (that are used for address computation in following basic blocks) in a table structure, which we later use for prefetch address generation. By making use of the run-time values of the architectural registers and with the help of the offset-based and loop-based enhancements, our prefetcher is capable of generating accurate and timely prefetches for data exhibiting both regular and irregular access patterns. It is to note that the branch-directed prefetcher does not need extra cache tag ports and it uses them only when they are idle. It is also implemented as a separate hardware entity and hence, it does not impact the main execution otherwise.

The current implementation of the branch-directed data prefetcher provides a mean benefit of 38.789 % over a system with no prefetching and 2.14 % over a system that implements the SMS prefetching for a set of 18 SPEC CPU2006 benchmarks. This improvement comes at a minimal additional hardware cost of 11.11 KB.

However as discussed in detail in the previous chapters, it is apparent that there is still significant scope of improvement with the Branch-Directed prefetching technique. We have observed that even though the prefetches issued by the branch-directed prefetcher are timely and accurate for most of the programs, the number of useless prefetches

generated is still high for some others. During the experiments, we had recorded significant performance improvements after incorporating the proposed prefetch-filtering mechanisms. Hence it is likely that even better performance gains can be achieved by exploring better filtering mechanisms. Techniques like dead-block prediction [27] can also be incorporated here to limit the degree of cache pollution, by prefetching into the predicted dead block positions only.

Also, branch-based correlation has been explored in the past mainly to realize instruction-prefetching. It is interesting to note that the branch Trace Cache structure used in our system can also be used to enable prefetching of instructions at branch target sites. Thereby, at no extra hardware cost a hybrid instruction and data prefetching solution can be realized, which may lead to further improvements in performance.

Though in the current thesis work, we have presented our solution for an inorder architecture, we would try to assess this prefetcher’s impact on other architectures as well like the superscalar or multithreaded ones in future.

REFERENCES

[1] A. J. Smith, “Cache memories,” *Computing Surveys*, vol. 14, no. 3, pp. 473-530, 1982.

[2] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. of the 8th Annual International Symposium on Computer Architecture*, Minneapolis, MN, May 12-14, 1981, pp. 81-87.

[3] T. R. Halfhill, "Intel's tiny atom," *Microprocessor Report*, Apr. 7, 2008, Available: www.mpronline.com.

[4] K. Krewell, "Sun’s Niagara pours on the cores," *Microprocessor Report*, Sep. 2004, Available: www.mpronline.com.

[5] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proc. of the 33rd Annual International Symposium on Computer Architecture*, Boston, MA, IEEE CS Press, Jun. 2006, pp. 252-263.

[6] M. Ferdman, S. Somogyi, and B. Falsafi, "Spatial memory streaming with rotated patterns," in *1st JILP Data Prefetching Championship*, Raleigh, NC, Feb. 2009.

[7] S. Kumar and C. Wilkerson, “Exploiting spatial locality in data caches using spatial footprints,” in *Proc. of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, Jun. 27–Jul. 1, 1998, pp. 357-368.

[8] D. Callahan, K. Kennedy, and A. Porterfield, “Software prefetching,” in *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, Apr. 1991, pp. 40-52.

[9] T. C. Mowry, M. S. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” in *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 1992, pp. 62-73.

[10] Y. Wu, “Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching,” in *Proc. of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, Jun. 2002, pp. 210-221.

[11] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Trans. on Computers*, vol. 11, no. 12, pp. 7-21, Dec. 1978.

[12] J. L. Baer and T. F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. of the 1991 ACM/IEEE Conference on Supercomputing*, Albuquerque, NM, Nov. 1991, pp. 176-186.

[13] R. Cooksey, S. Jordan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *Proc. of the 10th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002, pp. 279-290.

[14] E. Ebrahimi, O. Mutlu, and Y. Patt, “Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems,” in *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, Raleigh, NC, Feb. 2009, pp. 7-17.

[15] J. Dundas and T. Mudge, “Improving data cache performance by pre-executing instructions under a cache miss,” in *Proc. of the 11th International Conference on Supercomputing*, IEEE Press, Vienna, Austria, Jul. 1997, pp. 68-75.

[16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, “Runahead execution: An alternative to very large instruction windows for out-of-order processors,” in *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, Anaheim, CA, Feb. 2003, pp. 129-140.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, “Spatio-temporal memory streaming,” in *Proc. of the 36th Annual International Symposium on Computer Architecture*, Austin, TX, Jun. 2009, pp. 69-80.

[18] Y. Lui and D. R. Kaeli, "Branch-directed and stride-based data cache prefetching," in *Proc. of the International Conference on Computer Design*, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 255-230.

[19] S. Pinter and A. Yoaz, “Tango: A hardware-based data prefetching technique for superscalar processors,” in *Proc. of the 29th Annual International Symposium on Microarchitecture*, Paris, France, Dec. 1996, pp. 214–225.

[20] T. Chen and J. Baer, “Effective hardware based data prefetching for high-performance processors,” *IEEE Trans. on Computer Systems*, vol. 44, no. 5, pp. 609-623, May 1995.

[21] E. Jacobsen, E. Rotenberg, and J. E. Smith, “Assigning confidence to conditional branch predictions,” in *Proc. of the 29th Annual International Symposium on Microarchitecture*, Paris, France, Dec. 1996, pp. 142-152.

[22] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, “Confidence estimation for speculation control,” in *Proc. of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, Jun. 1998, pp. 122–131.

[23] D. A. Jimenez, “Composite confidence estimators for enhanced speculation control,” in *Proc. of the 21st International Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, Oct. 2009, pp. 161-168.

[24] K. Malik, M. Agarwal, V. Dhar, and M. I. Frank, “PaCo: Probability-based path confidence prediction,” in *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, Salt Lake City, UT, Feb. 2008, pp. 50-61.

[25] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 simulator: Modeling networked systems,” *Micro, IEEE*, vol. 26, no. 4, pp. 52-60, 2006.

[26] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer, "Set-dueling-controlled adaptive insertion for high-performance caching,” *Micro, IEEE*, vol. 28, no. 1, pp. 91-98, 2008.

[27] A. C. Lai, C. Fide, and B. Falsafi, “Dead-block prediction and dead-block correlating prefetchers,” in *Proc. of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Spain, Jul. 2001, pp. 144-154.

VITA

Reena Panda received her B.Tech. degree in electrical engineering from the National Institute of Technology (NIT) Rourkela, India in June 2008. After graduation, she worked as a design engineer at Samsung Electronics, India until July 2009. She entered Texas A&M University in August 2009 to pursue her master's degree in computer engineering and received her M.S in December 2011. Her research interests lie in the area of computer architecture.

Reena Panda may be reached at:
Department of Electrical and Computer Engineering
322 WERC,
Texas A&M University,
College Station, TX 77843-3128
e-mail: reena.panda@gmail.com